Parallelizing Hitting Set Enumeration Algorithms for Data Profiling

2024W 053021 LP Praktikum Informatik 1

Supervisor: Dr. Martin Schirneck

Department of Computer Science, University of Vienna

Author: Muhammad Raza Ali
Master's Student: 12248246, University of Vienna

14 February, 2025

Abstract

Minimal hitting set enumeration arises in a variety of data profiling tasks, particularly for discovering minimal unique column combinations in databases. In this report, we investigate whether the algorithm by Bläsius, Friedrich, Lischeid, Meeks, and Schirneck [JCSS'22] can be made significantly faster through parallelization. By distributing the search procedure across multiple compute nodes, we observe notable speedups on complex datasets.

1 Introduction

Minimal hitting set enumeration for data profiling is motivated by their significant applications, particularly in database optimization. A key question is whether the existing enumeration methods for discovering minimal unique column combinations (e.g. the one by *Bläsius, Friedrich, Lischeid, Meeks, and Schirneck* [1]) can be executed in a fraction of the time by harnessing parallel computation. Instead of running the enumeration sequentially on a single node, we explore a master–slave parallelism strategy that divides the workload among multiple nodes to reduce execution times substantially and computational complexities been tackled more efficiently.

1.1 Algorithmic Background

Hypergraph hitting set problems have long been studied for applications in combinatorics and database theory. In the specific context of data profiling, hitting sets capture the notion of *unique column combinations*, which are critical for tasks such as functional dependency detection and schema normalization. Recent work by Bläsius *et al.* [1] provides an efficient decision-tree approach with pruning (via an *extension oracle*) to systematically explore hitting sets while discarding branches that cannot yield minimal solutions. The question remains how well such an approach can *scale* when multiple processors are available, which is the main focus of this report.

1.2 Preliminaries

Hypergraphs: A *hypergraph* is a generalization of a standard graph in which an edge can contain any number of vertices (rather than exactly two). Formally, a hypergraph H is defined by a pair $(V, \mathcal{H})$ where V is the set of vertices and $\mathcal{H} \subseteq 2^V$ is the set of hyperedges.

For example, in **Figure 1**, the vertex set is $V = \{a, b, c, d, e\}$, and we have four hyperedges:

$$\{a, b, c\}, \quad \{a, b, c, d, e\}, \quad \{d, e\}, \quad \{b, c, d\}.$$

Unlike a standard (undirected) graph where each edge would just link two vertices, here an edge can link three, four, or even all five of the vertices at once.

Minimal Hitting Sets: A *hitting set* for $H = (V, E)$ is a subset $S \subseteq V$ that intersects every hyperedge $e \in E$. Equivalently, for each $e \in E$, we require $S \cap e \neq \emptyset$. A hitting set S is *minimal* if no proper subset of S is itself a hitting set. Removing any single element from a minimal hitting set would leave at least one hyperedge uncovered.

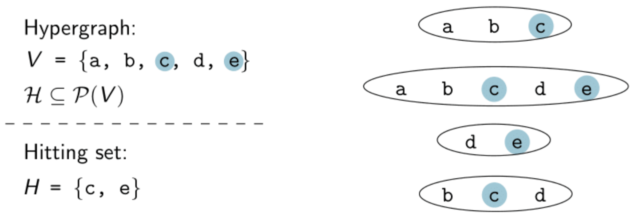

Figure 1: A hypergraph with vertices $\{a, b, c, d, e\}$. Each oval represents a hyperedge. The set $\{c, e\}$ hits every hyperedge and is minimal. (Figure by M. Schirneck.)

Age	Name	Address	City	Area Code
47	Mustermann, Max	Mittelstraße 125	Potsdam	D-14467
47	Mustermann, Max	W Broadway 400	San Diego	US-CA-92101
76	Doe, John	South Street 8	London	UK-W1K
90	Nightingale, Florence	South Street 8	London	UK-W1K
25	Menigmand, Morten	Trøjburgvej 24	Aarhus	DK-8200
33	Doe, John	South Street 8	Philadelphia	US-PA-19145

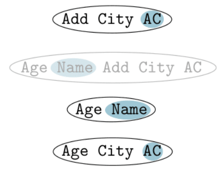

Figure 2: A table with attributes like `Age`, `Name`, `Address`, `City`, and `Area Code`, and its corresponding hypergraph representation. Potential unique column combinations map to minimal hitting sets. (Figure by M. Schirneck.)

Example. **Figure 1** shows a hypergraph with $V = \{a, b, c, d, e\}$ and four edges: $\{a, b, c\}$, $\{a, b, c, d, e\}$, $\{d, e\}$, $\{b, c, d\}$. The set $\{c, e\}$ is a minimal hitting set, as it intersects all edges, but neither $\{c\}$ nor $\{e\}$ alone is sufficient.

Unique Column Combinations in Databases: In database systems, a *unique column combination* (UCC) is a set of attributes (columns) such that every row in the table can be uniquely identified by at least one column in that set. **Figure 2** illustrates a small dataset with columns `Age`, `Name`, `Address`, `City`, and `Area Code`. If two rows share the same values for `Name` and `Age` (and have different values everywhere else), then $\{$`Name`, `Age`$\}$ alone would *not* uniquely distinguish those rows in general. But this pair shows that every UCC must contain either the `Address`, `City`, or `Area Code`. A proper subset of columns that fails to distinguish *at least one* pair of rows is useless as a UCC.

To reframe this as a hitting set problem, we note:

- **Vertices** correspond to column names.
- **Hyperedges** correspond to *difference sets*, which contain all attributes where the values in the two rows differ.

A minimal hitting set then represents a minimal set of columns that, collectively, breaks *every* collision, thus uniquely identifying each row. If we removed any column from this set, at least one pair of rows would remain indistinguishable. These examples demonstrate how fundamental notions of hypergraphs and hitting sets carry directly over to finding minimal UCCs.

1.3 Algorithm Overview

The goal is to compute all *minimal hitting sets*, ensuring that no subset of a solution is also a valid hitting set. A recursive decision tree-based approach is used to construct minimal hitting sets efficiently. The decision tree explores possible solutions by selecting elements iteratively, while an *extension oracle* prunes non-minimal or redundant branches. The extension oracle decides for a pair (X, Y) of disjoint sets of vertices, whether X can be extended to a minimal hitting set without using any vertex from Y.

Each branch in the decision tree corresponds to a candidate hitting set, and it *prunes non-minimal* or *redundant branches.*

The algorithm operates recursively as follows:

1. **Initialize** the decision tree with an empty set.
2. **Apply the extension oracle** to determine if the set is contained in a minimal hitting set.
3. **Expand** a set of elements that might form a hitting set iteratively into a decision tree to find minimal hitting sets.
4. **If minimal, store the hitting set**; if the set is not yet a minimal hitting set, but contained in one, then recourse and go back to Step 2 *Expand*; otherwise, prune the branch.
5. **Recurse on remaining elements** until all minimal hitting sets are found.

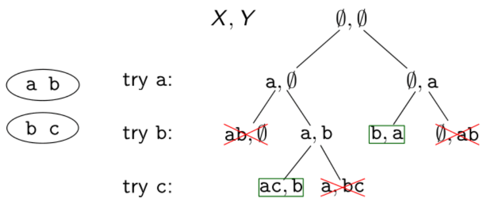

Figure 3: Decision tree for hitting set enumeration with extension oracle pruning. Green nodes indicate minimal solutions, while red nodes are pruned. (Figure by M. Schirneck.)

This recursive structure ensures that no redundant solutions are generated, as computing the extension oracle itself is very expensive. The tradeoff between pruning and the cost of the oracle is not always clear.

Base Case: The algorithm outputs X as a minimal hitting set when X intersects all hyperedges in Y, terminating recursion, with condition $X \cap e \neq \emptyset \, \forall e \in Y$, to the action (output and termination). **Recursive Expansion:** Each element in the vertex set V is selected one by one and added to X, forming a new candidate hitting set. **Pruning via Extension Oracle:** The function checks whether the newly constructed X remains minimal. **Backtracking:** If minimality fails, the branch is pruned, avoiding unnecessary computations.

2 Methodology

We base our work on an existing implementation of the hitting set enumeration algorithm in C++ by *Julius Lischeid* [4] . The implementation is run and tested on *ALMA* [2], a hardware resource of *University of Vienna*. ALMA is a "Heterogeneous Many-Core Cluster" comprising a frontend and multiple heterogeneous computational nodes, all based on Supermicro SuperServer systems. While the cluster itself has four computational nodes, our experiments utilize 16 virtual nodes created by dividing the physical nodes into multiple virtual instances. Each virtual node is connected via a network with dedicated memory. Transferring data from node to node is expensive in time (during program execution).

Parallelization Framework: We implement a master-slave architecture using the Message passing interface (MPI) to parallelize the hitting set enumeration algorithm. The design focuses on load-balanced task distribution while maintaining minimal communication overhead. The first node will be assigned as *master*, as involved in program execution (this can be any of the available nodes). The rest act as *slaves* or *workers*. We executed the algorithm on 16 nodes, with *one master* and *15 slaves*, and it is flexible enough to run with any number of nodes.

- **Master Node (1 node)**: It orchestrates workflow, manages the global task queue, allocates workers dynamically, dispatches subproblems (*branches*) to other nodes, and terminates the program when all tasks are exhausted and no worker is busy. It updates the *Tasks* and *Workers* list whenever a worker returns *right* branch data *(task)*, or completes a task, and assigns a task to available worker (free node).
  - **Task Queue**: Maintains pending tasks and available workers
  - **Non-blocking Communication**: Uses `MPI_Isend` for zero-copy task distribution
  - **Load Balancing**: Dynamically assigns tasks via checking worker availability, popping tasks from the queue, and streaming serialized data to workers.
  - **Termination Signal**: $(\texttt{TaskQueue.empty()} \wedge (\texttt{ActiveWorkers} = 0)$
- **Slave Nodes (15 nodes)**: They execute computational tasks via local branch exploration. Each worker processes assigned subproblems (partial hitting sets). If a branch is expandable, the left branch is computed locally, and the right branch data is sent back to the master. Workers notify the master upon task completion or when no further solutions exist (*pruning decision-making*).
  - **Asynchronous Receiving**: Non-blocking `MPI_Irecv` for task acquisition.
  - **Extendability Check**: Core pruning logic via the `extendable()` oracle:

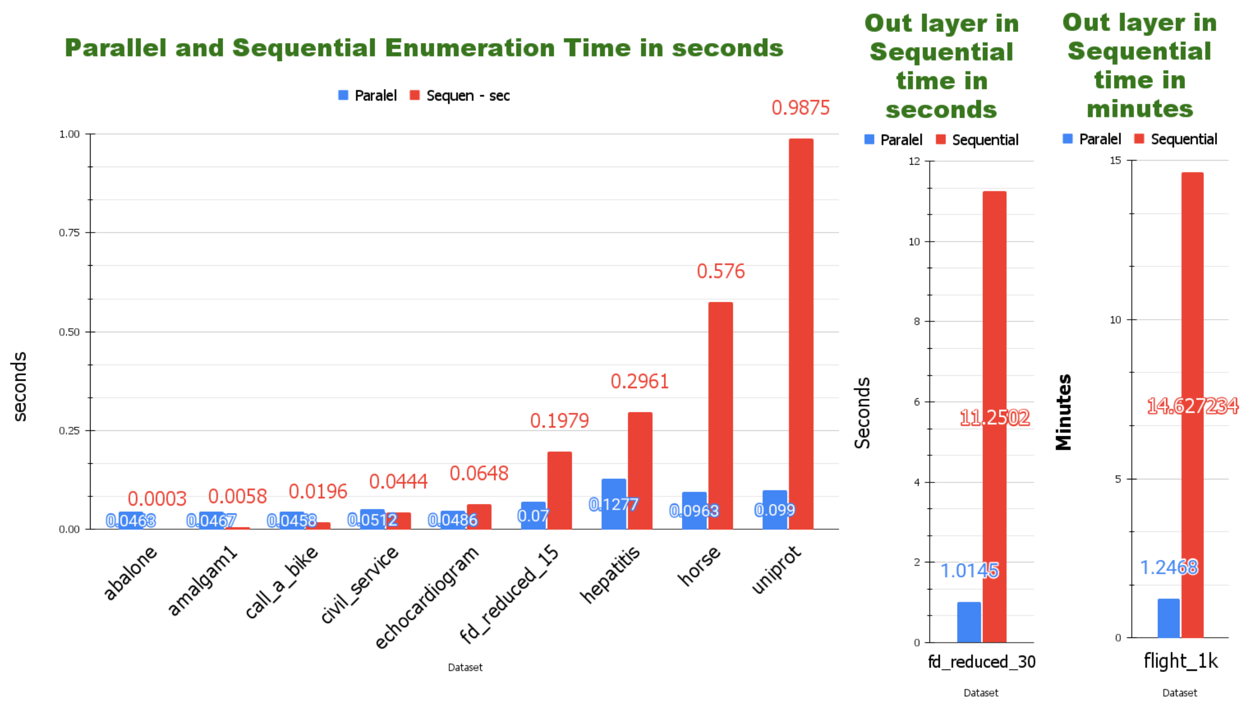

Figure 4: Comparison of parallel and sequential enumeration times on datasets. Parallel execution significantly reduces execution time, especially for larger datasets like flight_1k and fd_reduced_30, demonstrating its efficiency.

```
if (extendable(a, b)) {
    // Process left branch
    minimal_hitting_sets.push_back(a);
    // Return right branch via MPI_Isend
} else {
    // Prune branch and signal availability
}
```

- **Data Representation**: Data transfer between nodes may collect erroneous data without using the struct `WorkData`, mainly affecting the master node (possible to mix branches data). The master node simultaneously receives data from multiple nodes and assigns tasks. To mitigate this, we pack the entire branch data into a single struct, minimizing threading issues, as nodes lack shared memory for passing addresses. To encode hypergraphs, we serialize them as integer vectors. Our `WorkData` structure enables efficient task packaging.

```
struct WorkData {
    std::vector<int> x_vec, y_vec;        // Partial solution candidates
    int solution_size;                    // Serialized solutions count
    std::vector<int> flat_solutions;      // Compact solution representation
    unsigned long depth;                  // Recursion depth counter
};
```

3 Observations & Results

To evaluate the algorithm, we utilize eleven datasets from the *Hasso Plattner Institut* [3] and compare performance against the sequential enumeration method by *Bläsius et al.* [1]. We have direct access to the hypergraphs corresponding to the datasets and don't need to compute the translation from database to

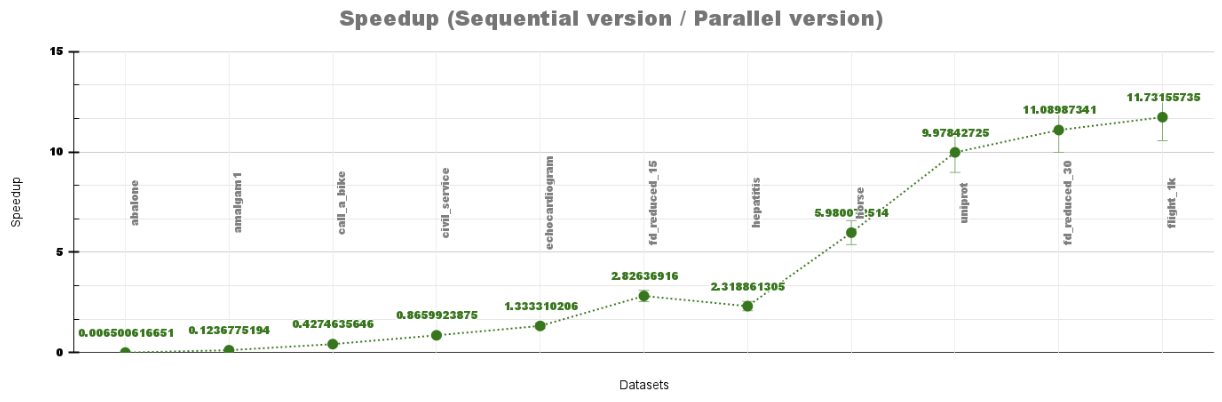

Figure 5: Speedup of parallel implementation over the sequential version across different datasets. The speedup is computed as the ratio of execution times (sequential/parallel).

hypergraph (which may take significant amount of time). **Figure 4** illustrates the performance of *ALMA* [2] across these datasets.

For smaller datasets (*abalone, amalgam1, call_a_bike,* and *civil_service*), the sequential method outperforms the parallel approach, with enumeration times of *0.0003, 0.0058, 0.0196,* and *0.0444* seconds (sequential) compares to *0.0463, 0.0467, 0.0458,* and *0.0512*seconds (parallel). This aligns with expectations, as parallelization overhead negates gains for trivial workloads.

For larger datasets (*echocardiogram, fd_reduced_15, hepatitis, horse, uniprot, fd_reduced_30,* and *flight_1k*), the parallel implementation achieves significant speedups. Parallel execution times (*0.0486, 0.07, 0.1277, 0.0963, 0.099, 1.0145* seconds, and *1.2468* minutes) are markedly lower than sequential times (*0.0648, 0.1979, 0.2961, 0.576, 0.9875, 11.2502* seconds, and *14.627234* minutes).

This concludes that the execution time of the `parallel` implementation is smaller compare to the `sequential` method for most datasets. The execution times vary significantly across the different instances, ranging from nanoseconds to minutes, depending on dataset size and search depth. However, the algorithm proves ineffective for datasets requiring enumeration times below `0.0648 seconds`, as the parallelization overhead negates performance gains.

We did some experiments with fewer than 16 nodes and observe that running on fewer nodes increases execution times due to reduced parallel efficiency. The methodology inherently prohibits single-node execution due to architectural constraints: designating a sole *master* node triggers an infinite loop when no *slave* nodes exist to receive distributed tasks. *Notably*, dual-node configurations exhibit higher execution times than the original sequential algorithm. This inefficiency stems from network communication costs, where the *master* must repeatedly transfer *workdata* to the *slave* node and receive processed *right-branch* results.

4 Performance Analysis

Figure 5 demonstrates varying degrees of speedup across datasets, with a maximum speedup of `11.73×` (`flight_1k`), corresponding to a `73.23%` reduction in execution time. The highlights the effectiveness of parallelization and the improved efficiency of hitting set enumeration. While some datasets exhibit speedup values below `1×` (e.g., `0.12×`), suggesting that parallelization introduced inefficiencies—likely due to the overhead of data transfer exceeding the time required for complete tree enumeration—others (`flight_1k`, `fd_reduced_30`, `uniprot`) achieve substantial acceleration, exceeding a `10×` improvement. The speedup is correlated with the sequential execution time.

5 Conclusion

Distributing the computation across multiple processors significantly reduces the enumeration time, particularly for complex datasets with large hypergraphs. By employing a master-slave strategy, parallelizing the hitting set enumeration algorithm achieves significant speedups for large datasets. However, smaller datasets suffer from parallelization overhead, leading to slowdowns.

Acknowledgements

I would like to express my gratitude to my supervisor, Martin Schirneck, for his invaluable guidance and support throughout this project. With his help, I was able to find an effective way to parallelize the algorithm and implementation with test datasets on ALMA [2]. His insights into the algorithm architecture significantly contributed to the design and implementation of our parallelization approach.

References

[1] T. Bläsius, T. Friedrich, J. Lischeid, K. Meeks, M. Schirneck, *Efficiently Enumerating Hitting Sets of Hypergraphs Arising in Data Profiling.* Journal of Computer and System Sciences, 2022 (previous version in ALENEX 2019). `https://arxiv.org/abs/1805.01310`

[2] ALMA, *Computer Science - Hardware Resources, University of Vienna.* `https://www.par.univie.ac.at/index.php?goto=hardware`.

[3] *Hasso-Plattner-Institut, Enumeration in Data Profiling* `https://hpi.de/friedrich/research/enumdat.html`

[4] *Special thanks go to Julius Lischeid C++ code on GitHub* `https://github.com/goodefroi/enumhyp`