arima

EPIsciences

Frequent Itemset-based Graph Numbering using Execution Traces for Cache Misses Reduction in Graph Analysis Tasks

Regis Audran MOGO WAFO[1], Thomas MESSI NGUELÉ[*1,2,3], Armel Jacques NZEKON NZEKO'O[*1,2], Xaveria Youh DJAM[1]

[1]University of yaounde 1, Department of Computer Science, PO:812, yaounde, Cameroon
[2]Sorbonne University, IRD, UMI 209 UMMISCO, F-93143, Bondy, France
[3]University of Ebolowa, HITLC, Computer Engineering Department, Cameroon

*E-mail : {audran.mogo, thomas.messi, armel.nzekon, xaviera.kimbi}@facsciences-uy1.cm

DOI : 10.46298/arima.18110
Submitted on 28 April 2026 - Published on 24 August 2026
Volume : **46** - Year : **2026**

Editors : *Mathieu Roche, Nabil Gmati, Moussa Lo*

Abstract

Graph analysis applications are increasingly used on large-scale datasets, where memory access patterns have a significant impact on performance. In such applications, poor data locality leads to a high number of cache misses, which in turn degrades execution time. To address this issue, several graph numbering techniques have been proposed to reorganize data in memory, mainly based on structural properties of the graph. More recently, execution trace-based approaches such as NumBaClus have been introduced, leveraging clustering techniques to group nodes with similar access patterns. However, these methods do not explicitly capture the co-occurrence of nodes within the same execution contexts. In this paper, we propose NumBaFrIt, a new graph numbering approach based on frequent itemset mining of execution traces. By modeling execution traces as transactional data, our method identifies groups of nodes that are frequently accessed together and reorganizes them in memory to improve data locality. Experimental results show that NumBaFrIt improves base numbering and significantly enhances existing ordering strategies when used as a preprocessing step, leading to better cache efficiency and reduced execution time. This is for example the case on user machine with 3072 KB of cache memory, our proposal got the best cache misses reduction through the combination NumBaFrit-sup70_cn-order with 48.96% compared to 17.15% of cache misses reduction gotten with Cn-order_cl-h (an existing numbering of NumBaClus).

Keywords

Frequent Itemset Mining, Cache Miss Reduction, Execution Traces, Graph Numbering

I INTRODUCTION

Graph analysis applications are widely used in many domains, including large-scale network analysis and scientific computing [6]. These applications typically involve iterative and irregular memory access patterns over large graphs, making memory performance a critical factor in overall execution efficiency.

As graph sizes continue to grow, execution time is increasingly dominated by memory access costs rather than pure computation [5]. In particular, frequent accesses to main memory, often caused by poor data locality, lead to a high number of cache misses and significantly degrade performance [4]. Improving memory access efficiency has therefore become a central challenge in graph processing. Since execution time is the most tangible performance metric, improvements in cache behavior generally translate into runtime reductions, although other architectural factors may influence the overall performance. Therefore, optimizing memory access patterns is essential not only for improving cache behavior but ultimately for reducing the execution time of graph analysis applications.

One effective approach to address this issue is to reorganize the layout of graph data in memory through node renumbering. The objective is to place related nodes close to each other in memory in order to improve cache utilization. Existing methods such as *cn-naive* [11], *Gorder* [9], *Rabbit-Order* [7], and *NumBaCo* [10] rely on structural properties of the graph, such as community structure or traversal patterns, to define node orderings. These approaches improve performance by enhancing spatial locality.

However, such methods do not explicitly take into account the actual runtime behavior of graph analysis applications. In practice, memory access patterns are strongly influenced by how the application interacts with the graph during execution [14].

To overcome this limitation, our previous work, **NumBaClus** [14], introduced a novel approach based on execution trace analysis. Instead of relying solely on the static structure of the graph, NumBaClus uses clustering techniques to group nodes with similar execution patterns. While this approach improves locality and reduces cache misses, it primarily captures similarity between execution traces and does not explicitly model the co-occurrence of nodes within the same execution context.

In this paper, we propose a new graph numbering method, **NumBaFrIt**, based on frequent itemset mining of execution traces. By modeling execution traces as transactional data, our approach identifies groups of nodes that are frequently accessed together during program execution. This allows us to explicitly capture co-access patterns and reorganize graph data in memory accordingly.

The experimental results show that NumBaFrIt not only improves base numbering but also significantly enhances existing ordering strategies when used as a preprocessing step. In particular, hybrid approaches combining NumBaFrIt with existing methods achieve the best performance in terms of execution time and cache efficiency.

The remainder of this paper is organized as follows. Section II presents the background through execution traces and memory locality. Section III reviews existing graph numbering approaches. Section IV presents the proposed NumBaFrIt model. Section V describes the experimental

framework and results. Section VI discusses the observed performance, and Section VII concludes the paper and outlines future work.

II BACKGROUND

In this section, we recall two essential concepts for understanding our contribution: execution trace presented in section 2.3 and memory locality presented in 2.4.

2.1 Graph analysis application

Graph analysis applications often exhibit irregular memory access patterns due to the sparse and non-contiguous nature of graph structures. In this work, we consider two representative graph analysis algorithms: PageRank and Betweenness Centrality.

2.1.1 PageRank

PageRank[2] is a graph analysis algorithm originally introduced to evaluate the importance of web pages according to the hyperlink structure of the Web. The fundamental idea is that a vertex is considered important if it is referenced by other important vertices.

The PageRank score of a vertex x is computed - iteratively as:

$$PR(x) = (1-d) + d \sum_{y \in N_{in}(x)} \frac{PR(y)}{|N_{out}(y)|} \tag{1}$$

where:

- d is the damping factor;
- $N_{in}(x)$ denotes the set of incoming neighbors of vertex x;
- $N_{out}(y)$ denotes the set of outgoing neighbors of vertex y.

The algorithm repeatedly traverses the graph until convergence is reached. Because each iteration requires accessing adjacency lists and neighboring vertices, PageRank generates a large number of memory accesses and is therefore particularly suitable for evaluating graph locality optimization techniques.

2.1.2 Betweenness Centrality

Betweenness Centrality (BC) [3] is a graph analysis metric that quantifies the importance of a vertex according to the number of shortest paths passing through it. A vertex with a high BC score acts as a bridge connecting different regions of the graph.

The Betweenness Centrality of a vertex v is defined as:

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

where:

- σ_{st} is the number of shortest paths between vertices s and t;
- $\sigma_{st}(v)$ is the number of those shortest paths that pass through vertex v.

In this work, the computation is performed using the algorithm proposed by Brandes , which significantly reduces the computational cost compared to a naive implementation.

Unlike PageRank, Betweenness Centrality relies on repeated shortest-path explorations and therefore exhibits different memory access patterns. Evaluating NumBaFrIt with both PageRank and Betweenness Centrality allows us to assess whether the proposed graph numbering generalizes across different classes of graph workloads.

2.2 Frequent Itemset Mining

Frequent Itemset Mining (FIM) [8] aims to discover groups of items that frequently appear together in a transactional dataset. In the context of NumBaFrIt, execution traces are transformed into transactions, and frequent itemsets represent groups of graph vertices that are repeatedly accessed together during graph analysis execution.

Several algorithms have been proposed for frequent itemset mining [8], including Apriori, FP-Growth, and Eclat. Although FP-Growth and Eclat are generally more efficient for very large transactional databases, Apriori was selected in this work for two reasons.

First, Apriori naturally generates frequent itemsets level by level according to their size. This property is particularly useful for NumBaFrIt because the renumbering algorithm processes itemsets progressively, starting from small itemsets and extending toward larger ones.

Second, Apriori provides explicit access to all intermediate frequent itemsets generated at each iteration. This allows us to directly exploit frequent itemsets of different sizes during the graph renumbering phase without requiring additional processing steps.

The worst-case complexity of Apriori remains exponential with respect to the number of distinct items. However, in our case, the support thresholds reduce the research space, making the mining phase tractable for the considered graph datasets.

2.3 Execution Traces

Execution traces provide a detailed view of how a graph analysis application accesses nodes during its execution. To collect these traces, we instrument the application to record sets of nodes that are accessed together during execution (see [14]).

Each line in the execution trace file represents a set of nodes accessed during a specific execution moment. This representation reveals co-access patterns between nodes and provides insight into the application's memory behavior.

The objective of our work is to exploit this information in order to construct a graph numbering in which nodes that are frequently accessed together are placed close to each other in memory.

2.4 Memory Access and Locality

Modern processors rely on hierarchical memory systems composed of multiple cache levels (L1, L2, and L3) to reduce the latency of memory accesses [4]. Accessing data from cache is significantly faster than accessing it from main memory.

The effectiveness of cache memory is largely based on the **principle of locality** (see on[5]), which states that memory accesses are not random but tend to follow predictable patterns. Two main types of locality are commonly observed:

- **Temporal locality**: data that have been accessed recently is likely to be accessed again in the near future.
- **Spatial locality**: data located near recently accessed memory locations is likely to be accessed soon.

These properties are critical for performance, as they allow cache systems to anticipate future accesses and reduce costly memory operations. Therefore, improving locality through graph reordering can significantly reduce cache misses and improve execution time.

In this context, graph numbering techniques aim to reorganize data in memory in order to better align with these locality principles. In fact, renumbering the graph contributes to give nodes that are frequently accessed together (during graph analysis execution) consecutive numbers, thus reducing cache misses and finally improving execution time.

III RELATED WORK: GRAPH NUMBERING

Numbering a graph consists of finding a new way to assign numbers to nodes in order to satisfy a performance property (cache references, cache misses or execution time). In this section, we provide an overview of the existing graph numbering approaches in two steps: Numbering for Graph Neural Network (GNN) and numbering for graph analysis applications. In the latter, we also discuss related work on structural and community-based analysis of complex networks, which, although not directly addressing graph numbering, offers complementary insights into the exploitation of structural properties considered in this work.

3.1 Numbering for Graph Neural Network (GNN)

Ruifeng Zhang et al. [15] propose a graph reordering algorithm that transforms a general irregular graph to required sparse patterns that can exploit the Sparse Tensor Cores (SPTC) incorporated in GPUs. It is then used to process GNNs (Graph Neural Networks).

Nikolai Merkel et al [13] show through the evaluation of 12 reordering strategies in two GNN systems (PyTorch Geometric and Deep Graph Library) that graph reordering is effective in reducing training time for CPU and GPU-based training, respectively. They also showed that GNN hyper-parameters influence the effectiveness of reordering and that lightweight reordering performs better for GPU-based training than for CPU-based training.

Here, the authors have proposed numbering systems designed to improve the training time of GNNs (Graph Neural Networks), whereas in this paper, we are more interested in graph analysis applications that run particularly on social networks. Furthermore, our goal in numbering is to reduce cache misses.

3.2 Complex Network Structure and Numbering for Classic Graph Analysis Applications

Graph numbering plays a key role in improving memory access efficiency in graph analysis applications. Early approaches in the literature mainly rely on the **structural properties of the graph**, such as community structure or traversal patterns, to define node orderings. Methods such as *cn-naive* [11], *Gorder* [9], *Rabbit-Order* [7], and *NumBaCo* [10] aim to place structurally related nodes close to each other in memory. These approaches implicitly improve memory performance by increasing **spatial locality**, that is, the likelihood that consecutively accessed data are stored in nearby memory locations.

More generally, the exploitation of structural and community properties has also been investigated in the analysis of complex networks. For instance, Gamgne Domgue et al. [12] proposed *UCAD* (commUnity disCovery method in Attribute-based multicoloreD networks), a hierarchical community discovery approach for attributed multicolored networks. UCAD introduces a node centrality measure called *attractivity* to identify the relevant dimensions of each node. These dimensions are then merged through a frequential aggregation into a single weighted graph, over which a Louvain-based optimization process groups nodes according to their relational information and their similarity on relevant dimensions, allowing the discovery of overlapping multi-communities. Although UCAD addresses community discovery rather than graph numbering, its consideration of structural and relational properties provides complementary insights into the structure-based graph reordering approaches considered in this work.

For graph numbering specifically, approaches such as *cn-naive*, *Gorder*, *Rabbit-Order*, and *NumBaCo* do not explicitly take into account the runtime behavior of graph analysis applications. In practice, memory performance is determined not only by the graph structure, but also by how the application accesses graph data during execution.

To address this limitation, our previous work introduced an approach based on **execution trace analysis**, called **NumBaClus** [14]. Instead of relying solely on the static structure of the graph, NumBaClus captures the dynamic behavior of graph analysis applications through execution traces.

3.3 NumBaClus Method

The NumBaClus approach leverages execution traces by applying clustering techniques to group nodes with similar access patterns.

First, a distance metric (defined in [14]) is used to quantify the similarity between nodes based on their execution traces. This distance measures how frequently nodes appear in similar execution contexts, allowing us to quantify their proximity. Nodes with smaller distances are more likely to be accessed together during execution.

Next, clustering algorithms such as *K-means* and *hierarchical clustering* are applied using this distance. Nodes that are considered close according to this metric are grouped into the same clusters.

Finally, nodes within each cluster are assigned consecutive identifiers, ensuring that they are placed close to each other in memory. This strategy improves memory access efficiency by increasing both spatial and temporal locality, leading to a reduction in cache misses and execution time.

3.4 Limitations and Motivation for NumBaFrIt

However, while NumBaClus captures similarity between execution traces, it does not explicitly model the **co-occurrence of nodes within the same execution trace**. As a result, nodes that are frequently co-accessed may not always be placed close to each other in memory. This limitation directly motivates the approach proposed in the section IV, where execution traces are analyzed through frequent itemset mining to explicitly capture co-access patterns between nodes.

IV NUMBERING WITH FREQUENT ITEMSETS

In this section, we introduce our graph numbering approach based on frequent itemset mining applied to execution traces. Unlike clustering-based methods, this approach explicitly models the co-occurrence of nodes within execution traces, enabling a more precise capture of memory access patterns.

The process consists of three main steps. First, execution traces are transformed into a transactional representation. Next, frequent itemsets are extracted to identify groups of nodes that are often accessed together. Finally, a reordering strategy is applied to assign consecutive identifiers to these nodes in order to improve memory locality.

4.1 Transforming Execution Traces

The main idea of this section is as follows: after extracting the execution traces of a graph analysis application, we obtain a dataset composed of multiple execution traces. Each trace is represented as a set of nodes accessed during the execution.

For example, execution traces can be represented as follows:

```
1: 2, 10, 11, 25, 22, 15, 3
2: 14, 10, 3, 6, 5, 20, 4
3: 1, 9, 50, 21, 6, 4, 12
...
```

In this representation: each **execution trace** corresponds to a **transaction**, while each **node** corresponds to an **item**. Thus, the execution trace file can be naturally modeled as a transactional dataset, where each transaction contains the set of nodes accessed together during a given execution trace.

The corresponding transactional dataset is:

```
2, 10, 11, 25, 22, 15, 3
14, 10, 3, 6, 5, 20, 4
1, 9, 50, 21, 6, 4, 12
...
```

This formulation allows us to directly leverage classical frequent itemset mining algorithms such as Apriori, without requiring additional transformation steps. Using this representation, we apply a frequent itemset mining algorithm to extract groups of nodes that are frequently accessed together. These groups, referred to as **Frequent Itemsets (FrIt)**, form the basis of our proposed numbering strategy.

Unlike clustering-based approaches that group nodes based on similarity of execution traces, our method focuses on co-occurrence patterns, capturing nodes that are frequently accessed together.

4.2 Frequent Itemset Extraction

To extract meaningful frequent itemsets, we rely on the notion of minimum support, which represents the minimum frequency at which a group of nodes must appear together across execution traces to be considered relevant.

In our approach, different support thresholds are explored. A higher support value leads to smaller but highly frequent groups of nodes, while a lower support allows capturing larger patterns that occur less frequently. In our experiments, support values such as 100, 75, 70, 65, and 60 were evaluated, as they provided a good trade-off between pattern significance and coverage.

This parameter directly influences the structure of the extracted itemsets and, consequently, the quality of the resulting node ordering in memory.

4.3 Reordering Strategy

The overall workflow of the proposed method is summarized in Figure 1. Starting from a raw graph, a graph analysis application (e.g., PageRank) is executed to generate execution traces. These traces are then transformed into a transactional dataset, from which frequent itemsets are extracted.

These itemsets capture co-occurrence patterns of nodes in execution traces. They are then used to guide the reordering of nodes in memory, placing frequently co-accessed nodes closer to each other. This improves spatial locality and increases the likelihood of temporal reuse, leading to better cache utilization and reduced cache misses.

4.4 NumBaFrIt Pipeline

Our proposed method, **NumBaFrIt**, leverages execution traces and frequent itemset mining to derive a memory-aware node ordering.

The process consists of the following key steps:

1. Execute a graph analysis application (e.g., PageRank) on the input graph G to collect execution traces (xtr)
2. The execution trace file (xtr) is transformed into a transactional dataset (T).
3. Use a frequent itemset mining algorithm (e.g., Apriori) to extract patterns from the execution traces, based on a minimum support threshold.
4. Apply a numbering strategy that assigns new node identifiers based on the extracted frequent itemsets.

The full pipeline is presented in Algorithm 1.

Algorithm 1 : **NumBaFrIt-order (Pipeline)**

Input: $G = (V, E)$, analysis application App, minimum support σ
Output: $G' = (V', E')$ (reordered graph)

1: $xtr \leftarrow Execute(G, App)$ {Run analysis application to collect execution traces}
2: $T \leftarrow TransformToTransactions(xtr)$ {Convert traces into transactional dataset}
3: $FrIt \leftarrow MineFrequentItemsets(T, \sigma)$ {Extract frequent itemsets with support $\geq \sigma$}
4: $G' \leftarrow NumberByItemsets(G, FrIt)$ {Reorder nodes based on FrIt}
5: **return** G'

Algorithm 2 : **Detailed Numbering Based on Frequent Itemsets**

Input: Frequent itemset file `File_Frequent_ItemSet.txt`
Output: Node renumbering array `tab_numerotation[]`

Read frequent itemsets from file into array `tab_itemset[]`
$max_size \leftarrow$ get the maximum itemset size from `tab_itemset[]`
$k \leftarrow 2$
$finished \leftarrow$ `false`
while $k \leq max_size$ **and** `not` $finished$ **do**
  for all node v in the current itemset **do**
    if v is not already numbered **then**
      Apply numbering to nodes in itemsets of size k from `tab_itemset[]`
    end if
  end for
  $finished \leftarrow$ check if all nodes have been numbered
  $k \leftarrow k + 1$
end while
Extract remaining unnumbered nodes from individual itemsets into `tab_item_indiv[]`

Sort `tab_item_indiv[]` by decreasing support
{Number the remaining nodes using `tab_item_indiv[]`}
for all node v in the current itemset **do**
  if v is not already numbered **then**
    assign next available identifier to v
  end if
end for
return `tab_numerotation[]`

At each step, nodes belonging to the same itemset are assigned consecutive identifiers, ensuring that they are placed close to each other in memory.

When processing a frequent itemset, only nodes that have not yet received a new identifier are renumbered. If a node already belongs to a previously processed itemset, its assigned identifier is preserved. This mechanism guarantees the consistency of the tab_numerotation array and prevents multiple assignments of the same node.

Algorithm 2 describes the detailed node numbering procedure based on frequent itemsets. The process begins by considering smaller itemsets (starting from size 2), under the assumption that strong pairwis relationships provide a reliable basis for building larger groups of frequently co-accessed nodes.

4.5 NumBaFrIt Complexity

The computational complexity of the *Mine Frequent Itemsets* step depends on the underlying Apriori algorithm [1]. Let T be the number of transactions generated from execution traces and I the number of distinct items.

Apriori proceeds level by level by generating candidate itemsets and scanning the transaction database to compute their support. Its complexity is therefore proportional to both the number of transactions and the number of generated candidates.

Let C denote the total number of candidate itemsets generated during the mining process. The complexity can be expressed as

$$\mathcal{O}(T \times C).$$

In the worst case, the number of candidates may grow exponentially with the number of distinct items, leading to

$$C = 2^{|I|} - 1$$

and therefore to a worst-case complexity of

$$\mathcal{O}(T \cdot 2^{|I|}).$$

However, Apriori exploits the downward-closure property to prune large portions of the search space.

In NumBaFrIt, execution traces exhibit strong locality patterns and relatively high minimum support thresholds are used. Consequently, only a limited number of candidates survive each pruning phase, which substantially reduces the practical mining cost.

The renumbering phase itself is substantially cheaper. Let N denote the number of extracted frequent itemsets, M the maximum itemset size, and V the number of graph vertices. The complexity of Algorithm 2 is approximately

$$\mathcal{O}(M^2 \cdot N + V \log V),$$

where the first term corresponds to scanning the frequent itemsets and the second term to sorting the remaining unnumbered vertices according to their support values.

Therefore, the dominant computational cost of NumBaFrIt lies in the frequent itemset mining stage, while the renumbering stage remains comparatively lightweight.

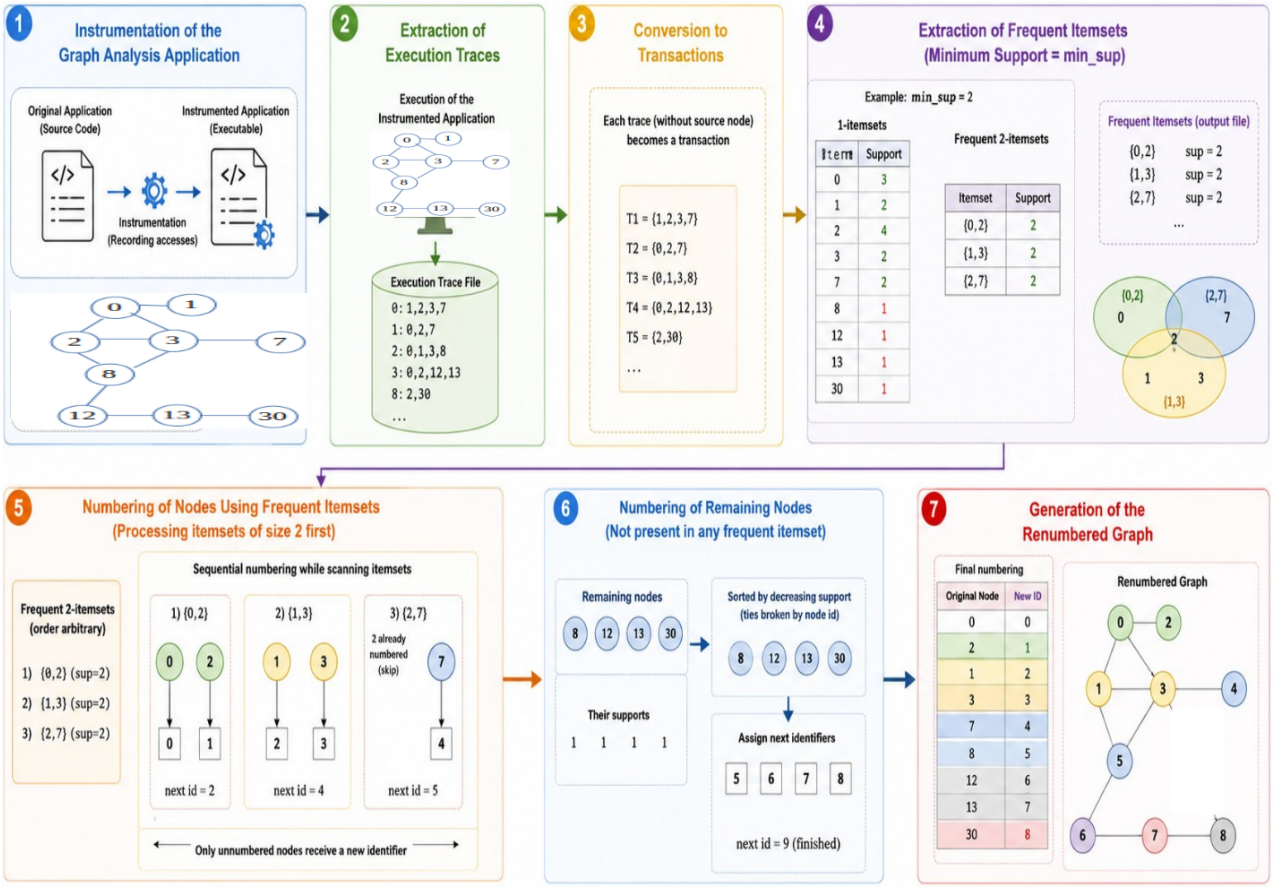

Figure 1: Principle of numbering using frequent itemsets

4.6 Illustrative Example of NumBaFrIt

To better explain the different stages of NumBaFrIt as seen in figure 1, we consider the following graph fragment represented as an edge list:

```
0 1
0 2
0 3
1 2
1 7
2 8
2 3
3 12
3 13
8 30
```

After instrumentation and execution of a graph analysis application, execution traces are generated. In our approach, each trace associates a source vertex with the vertices accessed during its processing. For example, the generated traces may be:

```
0:1,2,3,7
1:0,2,7
2:0,1,3,8
3:0,2,12,13
8:2,30
```

The next step consists in transforming these traces into transactions suitable for frequent itemset mining. The source vertex is removed and only the accessed vertices are retained. The resulting transactions are:

```
T1 = {1,2,3,7}
T2 = {0,2,7}
T3 = {0,1,3,8}
T4 = {0,2,12,13}
T5 = {2,30}
```

Assuming a minimum support threshold equal to 2, the support of each individual item is first computed.

Item	Support
0	3
1	2
2	4
3	2
7	2

Table 1: Frequent 1-itemsets

The frequent itemsets of size two are then extracted:

Frequent Itemset	Support
$\{0, 2\}$	2
$\{1, 3\}$	2
$\{2, 7\}$	2

Table 2: Frequent 2-itemsets

No frequent itemset of size greater than two satisfies the minimum support threshold. Consequently, the frequent itemset file generated by Apriori contains:

```
{0,2} sup=2
{1,3} sup=2
{2,7} sup=2
```

Algorithm 2 then applies the numbering process.

The first frequent itemset $\{0, 2\}$ receives consecutive identifiers:

$$0 \to 0, \qquad 2 \to 1$$

The second frequent itemset $\{1, 3\}$ is processed next:

$$1 \to 2, \qquad 3 \to 3$$

Finally, when processing the itemset $\{2, 7\}$, vertex 2 has already been numbered. Therefore, only vertex 7 receives a new identifier:

$$7 \to 4$$

This example illustrates an important property of NumBaFrIt: a vertex may belong to several frequent itemsets, but it receives a new identifier only once. Already numbered vertices are skipped during subsequent processing.

After processing all frequent itemsets, some vertices remain unnumbered because they do not belong to any frequent itemset satisfying the support threshold. In this example, these vertices are:

$$\{8, 12, 13, 30\}$$

These remaining vertices are stored in the `tab_item_indiv[]` structure, sorted according to their support values, and numbered afterwards:

$$8 \to 5, \qquad 12 \to 6, \qquad 13 \to 7, \qquad 30 \to 8$$

The final renumbering produced by NumBaFrIt is therefore:

Original Vertex	New Identifier
0	0
2	1
1	2
3	3
7	4
8	5
12	6
13	7
30	8

Table 3: Final numbering generated by NumBaFrIt

This example illustrates how NumBaFrIt groups together vertices that frequently appear in the same execution traces, while preserving a consistent numbering scheme for vertices belonging to multiple frequent itemsets and handling isolated vertices through the final support-based numbering phase.

V EXPERIMENTAL FRAMEWORK

The experimental environment we used to evaluate both scalability and algorithmic diversity is as follows.

- We perform experiments on two datasets from the SNAP collection [6]:
  - **Astro-ph** that contains 16,046 nodes and 242,502 edges
  - **Ca-CondMat** that contains 23,133 nodes and 93,497 edges
- We used two publicly available graph applications : the PageRank algorithm[1] and the Betweenness Centrality algorithm [2]. These two applications exhibit different computational and memory-access characteristics, enabling us to evaluate whether the proposed numbering method is generalizable beyond a single graph-analysis workload.
- Experiments were conducted on two user machines with the following specifications: the first computer: 4GB of RAM, Level 3 cache memory 3072 KB, L2 with 256KB and L1 with 32KB. The second computer: 16GB of RAM and Level 3 cache memory 4096 KB, L2 with 256KB and L1 with 32KB. We specify that the level 3 cache memory is shared between the cores for these two architectures

To assess the performance of different graph numbering methods, we use the following metrics: **Execution time** of the algorithms; **Cache misses** representing costly memory accesses beyond the last-level cache and **Cache references** indicating the total number of cache accesses during execution.

The purpose of this section is to assess the impact of NumBaFrIt on both execution performance and memory behavior. We first introduce the comparison strategies used to benchmark our approach against existing methods. Then, we analyze the experimental results in terms of execution time, cache misses, and cache references.

5.1 Comparison Strategies

Actually we can classify heuristic into three categories:

- **Existing Method**. In literature ([10], [14]) We observed that the community structure-based method that generally produced the best results was cn-order (see [11]) because it incorporated the advantages of the various other methods. Therefore, we chose Cn-order(cn-o) numbering to represent the community structure-based methods for graphs in our tables.
- **NumBaClus Method**. In our previous work, we have seen that in general, hierarchical Clustering order(cl-hier-order) and fusion between the cl_hier-order and cn-order produce the best performance and can represent NumBaClus method in Table 4 for 4cores machine with 4G Ram and Table 5 for 4cores machine with 16G Ram.
- **NumBaFrit Method**. To generate frequent itemsets from execution traces, we experimented with different minimum support thresholds. Among them, support values of 100, 75, 70, 65, and 60 produce several variants, among which the most significant improvements are observed in execution time, cache misses, and cache references.

[1] Used code is available here: https://github.com/nikos912000/parallel-pagerank
[2] Used code is available here: https://github.com/vineeths96/Betweenness-Centrality

To further validate the proposed numbering strategy, we extended our experiments beyond the original Astro-ph dataset and PageRank application. First, we evaluated PageRank on the larger Ca-CondMat graph in order to assess whether the observed gains remain valid on a different graph structure and size. Second, we considered another graph analysis application, namely Betweenness Centrality based on Brandes' algorithm. The corresponding results are reported in Table 6. This additional evaluation allows us to investigate whether the improvements provided by NumBaFrIt is generalizable to graph analysis tasks with different memory-access patterns.

For Tables 4, 5 and 6, best performance for NumBaFrit method have specific colors:

- For reduction of execution time, best improvement of NumBaFrit take **blue** color;
- For cache misses reduction, best improvement of NumBaFrit take **red** color;
- For cache reference reduction, best improvement of NumBaFrit take **orange** color

Heuristic	Time (ms)	Cache misses (th)	Cache references (th)
Astro-ph 1th	977.60	1217.29	28980.58
Cn-n-o 1th	939.39 (3.91%)	1064.52 (12.55%)	14751.92 (49.10%)
cl-h-o 1th	953.78 (2.44%)	1240.05 (-1.87%)	22475.02 (22.45%)
Cn-o_cl-h-o 1th	901.94 (7.74%)	838.97 (31.08%)	15993.70 (44.81%)
NumBaFrIt-sup60-o 1th	962.41 (1.55%)	1027.47 (15.59%)	28240.23 (2.55%)
NumBaFrIt-sup65-o 1th	959.50(1.85%)	900.70 (26.01%)	28037.71 (3.25%)
NumBaFrIt-sup70_cn-n-o 1th	**889.77 (8.99%)**	**790.07 (35.10%)**	**14517.26 (49.91%)**
NumBaFrIt-sup65_cl-h-o 1th	910.77 (6.84%)	861.65 (29.22%)	16566.78 (42.83%)
NumBaFrIt-sup60_Cn-o_cl-h-o 1th	**890.89 (8.87%)**	**720.12 (40.84%)**	**14662.22 (49.41%)**
Astro-ph 3th	779.86	1973.98	33197.75
Cn-n-o 3th	772.38 (0.96%)	2342.67 (25.77%)	19085.18 (42.51%)
cl-h-o 3th	814.84 (-4.49%)	2072.40 (-4.99%)	26466.54 (20.28%)
Cn-n-o_cl-h-o 3th	730.19 (6.37%)	1635.43 (17.15%)	19489.40 (41.29%)
NumBaFrIt-sup60-o 3th	879.44 (-12.77%)	1916.74 (2.90%)	32708.21 (1.47%)
NumBaFrIt-sup65-o 3th	872.24 (-11.85%)	2053.53 (-4.03%)	32672.20 (1.58%)
NumBaFrIt-sup70_cn-n-o 3th	**717.89 (7.95%)**	**1610.74 (48.96%)**	**18438.81 (44.46%)**
NumBaFrIt-sup65_cl-h-o 3th	736.51 (5.56%)	1910.01 (3.24%)	19948.69 (39.91%)
NumBaFrIt-sup60_Cn-n-o_cl-h-o 3th	**729.03 (6.52%)**	**1594.13 (19.24%)**	**19419.33 (41.50%)**
Ca-CondMat 1th	1596.61	5167.87	56995.69
Ca-CondMat_FrIt-sup60-o 1th	1525.59 (4.45%)	4545.34 (12.05%)	47000.33 (17.54%)
Ca-CondMat_FrIt-sup65-o 1th	1520.61 (4.76%)	4217.23 (18.40%)	46913.51 (17.69%)
Ca-CondMat_FrIt-sup70-o 1th	1523.39 (4.59%)	4434.94 (14.18%)	46885.78 (17.74%)
Ca-CondMat_FrIt-sup75-o 1th	1526.40 (4.40%)	4530.25 (12.34%)	46936.33 (17.65%)
Ca-CondMat_FrIt-sup100-o 1th	1523.95 (4.55%)	4543.64 (12.08%)	47107.39 (17.35%)
Ca-CondMat 3th	1287.58	6780.56	62508.81
Ca-CondMat_FrIt-sup60-o 3th	1247.47 (3.12%)	6091.20 (10.17%)	52387.72 (16.19%)
Ca-CondMat_FrIt-sup65-o 3th	1251.81 (2.78%)	6044.46 (10.86%)	52260.49 (16.40%)
Ca-CondMat_FrIt-sup70-o 3th	1238.50 (3.81%)	6197.56 (8.60%)	52329.88 (16.28%)
Ca-CondMat_FrIt-sup75-o 3th	1242.53 (3.50%)	6269.53 (7.54%)	52334.51 (16.28%)
Ca-CondMat_FrIt-sup100-o 3th	1233.69 (4.19%)	6066.03 (10.54%)	52382.50 (16.20%)

Table 4: Graph ordering comparison on first user machine with PageRank (time in ms, cache misses and references in thousands)

5.2 Observation on Execution Time Reduction

In this section, we analyze the execution time reduction using Tables 4 and 5. As shown in our previous work ([14]), execution time reduction reflects the gain obtained through graph reordering when executing PageRank on graph datasets. In addition to the Astro-ph graph used in our previous experiments, we also evaluate the proposed approach on the larger Ca-CondMat graph in order to assess its effectiveness on a different graph structure and scale.

These improvements are directly related to the reduction of cache references and cache misses, discussed respectively in Sections 5.4 and 5.3.

In Table 4, we observe that:

- With one thread, NumBaFrIt-based approaches consistently improve the baseline ordering. In particular, *NumBaFrIt-sup70_cn-order* achieves the best performance with an execution time reduction of **8.99%**, outperforming all other methods. Similarly, *NumBaFrIt-sup60_Cn-o_cl-hier-order* also provides a significant gain of **8.87%**. These results show that combining NumBaFrIt with existing heuristics leads to better performance than using the original methods alone.
- With three threads, the same trend is observed. The hybrid approaches again provide the best results, with *NumBaFrIt-sup70_cn-order* achieving the highest improvement of **7.95%**, followed by *NumBaFrIt-sup60_Cn-o_cl-hier-order* with **6.52%**. In contrast, standalone NumBaFrIt configurations (sup60-o and sup65-o) show negative gains, highlighting the importance of combining NumBaFrIt with existing ordering strategies.

To further address scalability concerns, we also evaluated PageRank on the larger Ca-CondMat dataset containing 23,133 vertices and 93,497 edges. The results reported in Table 4 show that NumBaFrIt remains effective on this graph. For a single-thread execution, the best configuration (NumBaFrIt-sup65-o) reduces execution time by 4.76%. Similar improvements are observed with three threads, where NumBaFrIt-sup100-o achieves the highest execution-time reduction of 4.19%.

In contrast to Astro-ph, all NumBaFrIt configurations produce positive execution-time gains on the larger Ca-CondMat graph in the three-thread configuration. This difference can be explained by the larger memory footprint of Ca-CondMat, which exceeds the cache capacity of the experimental machines and therefore makes execution more sensitive to memory locality and then to renumbering. As a result, the benefits introduced by NumBaFrIt become more visible, whereas on Astro-ph the graph can largely fit into cache and the impact of numbering is consequently reduced. This observation also helps explain why some standalone NumBaFrIt configurations exhibit negative gains on Astro-ph while achieving consistent improvements on Ca-CondMat.

These results confirm that the benefits of NumBaFrIt are not restricted to the Astro-ph dataset. Although the magnitude of the gains varies according to graph structure and execution configuration, the proposed numbering continues to provide measurable execution-time improvements on a different graph dataset.

In Table 5, we observe that:

- With one thread, all NumBaFrIt-based combinations improve the baseline ordering. The best performance is achieved by *NumBaFrIt-sup100_Cn-o_cl-hier-order*, with a reduction of **20.22%**, followed by *NumBaFrIt-sup70_cl-hier-order* with **11.89%**. In addition, *NumBaFrIt-sup75_cn-order* also significantly improves execution time (**10.33%**), confirming the effectiveness of combining NumBaFrIt with existing heuristics.
- With three threads, the improvements remain consistent. The best result is obtained with *NumBaFrIt-sup75_cn-order*, achieving a reduction of **14.59%**, followed by *NumBaFrIt-sup100_Cn-o_cl-hier-order* with **9.01%**. These results confirm that integrating NumBaFrIt into existing ordering strategies leads to significant performance gains, especially in multi-threaded settings.

In summary, these results confirm that NumBaFrIt can significantly improve execution time under specific conditions, particularly when combined with existing ordering heuristics. While standalone configurations do not always lead to performance gains, hybrid approaches consistently outperform traditional methods. This highlights the effectiveness of NumBaFrIt as a complementary optimization technique rather than a replacement for existing strategies. We now turn to the analysis of its impact on cache misses.

5.3 Observation on Cache Misses Reduction

In our previous work (see [14]), cache misses correspond to the number of times the processor requests data from the last-level cache (L3) but fails to find it, forcing access to main memory. Reducing cache misses is therefore crucial, as it directly impacts memory latency and overall execution performance.

The two experimental machines used in this work have different last-level cache capacities: 3072 KB for the first machine and 4096 KB for the second one. For the PageRank experiments, the Astro-ph graph occupies approximately 2.46 MB, whereas the larger Ca-CondMat graph occupies approximately 4.1 MB after graph preparation. Consequently, Astro-ph can fit almost entirely within the largest cache configuration, while Ca-CondMat exceeds the cache capacity of both machines and therefore generates a higher pressure on the memory hierarchy.

From Tables 4 and 5, we observe that the proposed NumBaFrIt-based heuristics can significantly reduce cache misses, although the impact varies depending on the configuration.

In Table 4, we observe that:

- With one thread, hybrid approaches combining NumBaFrIt with existing heuristics provide the best reductions. In particular, *NumBaFrIt-sup60_Cn-o_cl-hier-order* achieves a reduction of **40.84%**, while *NumBaFrIt-sup70_cn-order* reaches **35.10%**. These results clearly outperform both the baseline and the standalone heuristics.
- With three threads, the same trend is observed. The best performance is achieved by *NumBaFrIt-sup70_cn-order*, with a reduction of **48.96%**, followed by *NumBaFrIt-sup60_Cn-o_cl-hier-order* with **19.24%**. In contrast, standalone NumBaFrIt configurations show limited or even negative improvements, confirming that the benefit mainly comes from combining strategies.

To further investigate the impact of graph size on memory behavior, we also evaluated PageRank on the larger Ca-CondMat dataset. The results reported in Table 4 show that NumBaFrIt remains effective on this graph and produces noticeable reductions in cache misses.

For single-thread execution, the best configuration (*NumBaFrIt-sup65-o*) achieves a cache miss reduction of **18.40%**. For three-thread execution, the same configuration provides a reduction of **10.86%**. These additional results indicate that the proposed numbering strategy improves memory locality on different graph structures and remains beneficial when the graph size exceeds the last-level cache capacity.

In Table 5, we observe that:

- With one thread, with NumBaFrIt-based the improvements are generally lower than those observed on the Dell machine, which can be explained by the larger cache size.

- With three threads, the improvements become more moderate. Standalone NumBaFrIt configurations achieve only slight reductions (around **1–2%**), while hybrid approaches still provide consistent but less pronounced gains. This confirms that when the cache is large enough to hold most of the graph, the margin for reducing cache misses is naturally reduced.

Overall, these results show that NumBaFrIt is effective in reducing cache misses, especially when combined with existing ordering strategies. However, its impact is strongly influenced by hardware characteristics, particularly the cache size relative to the graph size. We now proceed to analyze the impact of these reorderings on cache references.

Heuristic	**Time (ms)**	**Cache misses (th)**	**Cache references (th)**
Astro-ph 1th	1253.22	1826.24	107621.75
Cn-n-o 1th	1131.63 (9.70%)	2099.03 (-14.94%)	60462.00 (43.82%)
cl-h-o 1th	1218.75 (2.75%)	3679.01 (-11.70%)	95998.16 (22.37%)
Cn-n-o_cl-h-o 1th	1158.27 (7.58%)	2384.75 (-30.58%)	64171.74 (40.37%)
NumBaFrIt-sup60-o 1th	1216.99 (2.89%)	1918.76 (-5.07%)	109408.09 (-1.66%)
NumBaFrIt-sup75-o 1th	1186.77 (5.30%)	1892.76 (-3.64%)	107475.08 (0.14%)
NumBaFrIt-sup75_cn-n-o 1th	1123.81 (10.33%)	2317.60 (-26.91%)	**60548.65 (43.74%)**
NumBaFrIt-sup70 _cl-h-o 1th	**1104.16 (11.89%)**	2311.35 (-26.56%)	65438.19 (39.20%)
NumBaFrIt-sup100_Cn-o_cl-h-o 1th	**999.85 (20.22%)**	2242.88 (-22.81%)	**63930.06 (40.60%)**
Astro-ph 3th	950.12	5492.86	128031.09
Cn-n-o 3th	842.92 (11.28%)	5685.05 (-3.50%)	79500.43 (37.91%)
cl-h-o 3th	977.27 (-2.86%)	6782.76 (-16.96%)	110576.47 (18.92%)
Cn-n-o_cl-h-o 3th	938.92 (1.18%)	6019.37 (-9.59%)	83117.71 (35.08%)
NumBaFrIt-sup60-o 3th	1087.45 (-14.45%)	**5369.25 (2.25%)**	128304.63 (-0.21%)
NumBaFrIt-sup75-o 3th	1141.98 (-20.19%)	**5343.36 (2.72%)**	129587.65 (-1.22%)
NumBaFrIt-sup75_cn-n-o 3th	**811.54 (14.59%)**	5971.17 (-8.71%)	**80276.24 (37.30%)**
NumBaFrIt-sup70 _cl-h-o 3th	949.45 (0.07%)	5424.47 (1.25%)	84061.15 (34.34%)
NumBaFrIt-sup100_Cn-o_cl-h-o 3th	**864.48 (9.01%)**	5788.27 (-5.38%)	**83113.30 (35.08%)**

Table 5: Graph ordering comparison on second user machine(time in ms, cache misses and references in thousands)

5.4 Observation on Cache References Reduction

In our previous work (see [14]), cache references correspond to the total number of accesses made by the processor to the cache hierarchy (L1, L2, and L3) and specialy for this case in level 1 and 2 of cache memory. Reducing cache references therefore reflects an improvement in memory locality, as it indicates that data is reused more efficiently and fewer redundant memory accesses are required.

From Tables 4 and 5, we observe that NumBaFrIt-based heuristics significantly impact cache reference behavior, especially when combined with existing ordering strategies.

In Table 4, we observe that:

- With one thread, hybrid approaches provide the most significant reductions. In particular, *NumBaFrIt-sup70_cn-order* achieves the best result with a reduction of **49.91%**, closely followed by *NumBaFrIt-sup60_Cn-o_cl-hier-order* with **49.41%**. These results clearly outperform both the baseline and standalone heuristics, and are consistent with the strong improvements observed in execution time.
- With three threads, the same trend is maintained. The best performance is again achieved by *NumBaFrIt-sup70_cn-order* with **44.46%**, followed by *NumBaFrIt-sup60_Cn-o_cl-hier-order* with **41.50%**. These reductions confirm that improving data locality through graph reordering remains effective even in parallel settings.

To further evaluate the effect of NumBaFrIt on memory accesses, we also analyzed the larger Ca-CondMat dataset using PageRank. The results reported in Table 4 show that the proposed numbering continues to reduce cache references across different graph structures.

For single-thread execution, the best configuration (*NumBaFrIt-sup70-o*) achieves a reduction of **17.74%** in cache references, while *NumBaFrIt-sup65-o* achieves a very similar reduction of **17.69%**. For three-thread execution, the largest reduction is obtained with *NumBaFrIt-sup65-o*, which decreases cache references by **16.40%**. These results confirm that NumBaFrIt improves data locality by reducing the total number of cache accesses required during graph processing.

Compared with the reductions observed on Astro-ph, the improvements on Ca-CondMat are smaller in magnitude but remain substantial. This behavior is expected because the effectiveness of graph reordering depends not only on graph size but also on structural properties such as connectivity patterns and vertex neighborhood organization. Nevertheless, the consistent reduction of cache references across both datasets indicates that the proposed numbering strategy is able to improve memory locality on different graph topologies.

In Table 5, we observe that:

- With one thread, NumBaFrIt-based combinations again provide strong improvements. The best result is obtained by *NumBaFrIt-sup75_cn-order* with a reduction of **43.74%**, while *NumBaFrIt-sup100_Cn-o_cl-hier-order* achieves **40.60%**. These results demonstrate that the proposed method effectively reduces unnecessary cache accesses.
- With three threads, the improvements remain significant but slightly reduced compared to the one-thread case. The best performance is achieved by *NumBaFrIt-sup75_cn-order* with **37.30%**, followed by *NumBaFrIt-sup100_Cn-o_cl-hier-order* with **35.08%**. This behavior is consistent with the reduced impact of locality optimizations in multi-threaded environments.

Overall, these results show that NumBaFrIt significantly improves memory locality by reducing cache references, especially when combined with existing heuristics. This reduction is directly correlated with the improvements observed in execution time, confirming the effectiveness of the proposed approach. In the next section, we synthesize these observations to provide a global analysis of the performance gains.

5.5 Observations on Executions with Betweenness Centrality

Heuristic	Time (s)	Cache misses (hTh)	Cache references (hTh)
Ca-CondMat 1th	351.70	44126.63	121745.25
NumBaFrIt-sup60-o 1th	326.70 (7.11%)	42603.74 (3.45%)	114791.43 (5.71%)
NumBaFrIt-sup65-o 1th	332.04 (5.59%)	41736.57 (5.42%)	115177.60 (5.39%)
NumBaFrIt-sup70-o 1th	332.40 (5.49%)	43782.08 (0.78%)	114557.71 (5.90%)
NumBaFrIt-sup75-o 1th	328.95 (6.47%)	41824.64 (5.22%)	115034.79 (5.51%)
NumBaFrIt-sup100-o 1th	330.64 (5.99%)	41593.41 (5.74%)	114900.23 (5.62%)
Astro 1th	258.37	30395.05	71210.84
Astro_FrIt-sup70-o 1th	257.10 (0.49%)	31634.14 (-4.08%)	70887.74 (0.45%)

Table 6: Graph ordering comparison on first user machine with Betweenness Centrality(time in ms, cache misses and references in thousands)

Table 6 reports the execution of Betweenness Centrality on the Ca-CondMat dataset using the original numbering and several NumBaFrIt-generated orderings. The results show that the best NumBaFrIt configurations improve the baseline graph representation. The largest observed gains reach **7.11%** in execution time, **5.74%** in cache misses, and **5.90%** in cache references.

These improvements are particularly significant because Betweenness Centrality exhibits memory-access patterns that differ substantially from those of PageRank. Consequently, the results provide evidence that the benefits of NumBaFrIt are not limited to a single graph-analysis workload.

To further investigate this aspect, we also executed Betweenness Centrality on the Astro-ph dataset. As shown in Table 6, the NumBaFrIt-sup70-o ordering still achieves a modest execution-time improvement of 0.49% together with a 0.45% reduction in cache references. Although the gains are smaller than those observed on Ca-CondMat, they indicate that the proposed numbering can remain beneficial across different graph structures and analysis tasks.

VI DISCUSSION

In this section, we analyze the impact of NumBaFrIt on both base and existing graph ordering strategies.

6.1 NumBaFrIt vs Base Numbering

The results obtained by NumBaFrIt are based on an upstream analysis of execution traces using frequent itemset mining with different support thresholds (100, 75, 70, 65, and 60). These configurations generate several graph orderings, providing flexibility according to the characteristics of the graph and the target execution platform.

From the experimental results on the Astro-ph dataset, we observe that several standalone NumBaFrIt variants, such as *NumBaFrIt-sup60-o*, *NumBaFrIt-sup65-o*, and *NumBaFrIt-sup75-o*, are able to improve the base numbering in single-thread execution. These variants achieve measurable reductions in execution time while also reducing cache misses and cache references, indicating improved memory locality.

The additional experiments performed on the larger Ca-CondMat dataset confirm this tendency. For PageRank executions, all tested NumBaFrIt configurations improve the baseline numbering in the single-thread configuration, with the best execution-time reduction reaching **4.76%**. Similar improvements are also observed in cache behavior, with reductions of up to **18.40%** in cache misses and **17.69%** in cache references. Unlike Astro-ph, positive execution-time gains are also consistently observed in the three-thread configuration, suggesting that the benefits of NumBaFrIt become more visible when the graph places greater pressure on the memory hierarchy.

To further evaluate the generality of the approach, we also executed Betweenness Centrality on both Astro-ph and Ca-CondMat. On Ca-CondMat, the best NumBaFrIt configuration achieves a reduction of **7.11%** in execution time, together with reductions of **5.74%** in cache misses and **5.90%** in cache references. Although the gains are more modest on Astro-ph, the results remain positive, demonstrating that NumBaFrIt is not restricted to PageRank workloads.

Overall, these observations indicate that the proposed numbering strategy is able to improve memory locality across different graph datasets and graph analysis applications. The magnitude of the gains depends on graph topology, graph size, cache capacity, and execution configuration, but the results consistently show that exploiting frequent co-access patterns extracted from execution traces provides a more efficient node layout than the original graph numbering.

6.2 Improving Existing Methods with NumBaFrIt

Here, we evaluate how NumBaFrIt enhances our previous work, particularly NumBaClus, by acting as a preprocessing step before applying existing ordering heuristics. NumBaFrIt introduces a support-based parameterization that allows multiple variants of the method. This flexibility enables the method to adapt to different execution environments and workloads.

The experimental results show that combining NumBaFrIt with existing heuristics (such as *cn-order* and *cl-hier-order*) consistently leads to better performance than using these methods alone. For instance, hybrid approaches such as *NumBaFrIt-sup70_cn-order* and *NumBaFrIt-sup60_Cn-o_cl-hier-order* outperform their standalone counterparts in both execution time and cache behavior.

These improvements can be explained by the ability of NumBaFrIt to enhance spatial locality by reorganizing graph vertices so that frequently accessed elements are stored close to each other in memory. This allows multiple useful data elements to be loaded within a single cache line. Additionally, it enhances temporal locality by increasing the likelihood that recently accessed data will be reused before being evicted from the cache. As a result, subsequent ordering methods operate on data layouts that are already optimized for memory access, leading to fewer cache misses and more efficient cache usage.

NumBaFrIt should primarily be viewed as a preprocessing technique. The objective is not to replace existing graph-ordering methods, but rather to provide an initial numbering that exposes execution-related locality information before a more specialized reordering algorithm is applied.

The frequent itemsets extracted from execution traces capture groups of vertices that are repeatedly accessed together during graph processing. By placing these vertices closer in memory, NumBaFrIt produces an initial layout that already reflects part of the application's access behavior. Community-based methods can then further refine this layout using structural properties of the graph.

The experimental results support this interpretation. Across several configurations, the best performance is often obtained when NumBaFrIt is combined with existing community-based orderings rather than used alone. This suggests that execution-driven locality information and graph-structure information are complementary. NumBaFrIt provides a locality-aware starting point, while community-based methods further exploit the underlying graph organization.

More generally, both NumBaFrIt and community-based reorderings can be considered preprocessing stages whose objective is to improve the memory layout of the graph before executing graph analytics applications.

VII CONCLUSION

In this paper, we proposed a new graph numbering approach based on the exploitation of execution traces through frequent itemset mining. In this model, graph nodes are considered as items, while execution traces are treated as transactions. This allows us to identify groups of nodes that are frequently accessed together during program execution, leading to the design of the NumBaFrIt model.

This work is a natural continuation of our previous approach, NumBaClus, which also relied on execution trace analysis but used clustering techniques to group nodes with similar execution patterns. However, NumBaClus primarily focused on similarity between execution traces, without explicitly capturing the co-occurrence of nodes within the same execution traces.

NumBaFrIt addresses this limitation by directly modeling execution traces as itemsets, thus enabling the identification of nodes that are frequently accessed together. This results in a more precise reorganization of graph data in memory, improving both spatial and temporal locality.

The experimental results show that NumBaFrIt is able to improve base numbering and, more importantly, significantly enhance existing ordering strategies when used as a pre-processing step. In particular, hybrid approaches combining NumBaFrIt with methods such as *cn-order* and *cl-hier-order* achieve the best overall performance in terms of execution time and cache efficiency.

Rather than introducing a completely new direction, this work confirms the relevance of leveraging execution trace analysis for graph data layout optimization. It also demonstrates that combining data mining techniques, such as frequent itemset mining, with existing ordering strategies can lead to substantial performance improvements.

As future work, we plan to further investigate the interplay between structure-based and execution trace-based numbering strategies, with the objective of maximizing execution performance through improved memory locality.

Publications

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. "Fast discovery of association rules". In: *Advances in Knowledge Discovery and Data Mining*. Edited by U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. AAAI/MIT Press, 1996, pages 307–328.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank citation ranking: Bringing order to the web*. Technical report 1999-66. Stanford InfoLab, 1999.

[3] U. Brandes. "A Faster Algorithm for Betweenness Centrality". In: *Journal of Mathematical Sociology* 25.2 (2001), pages 163–177.

[4] U. Drepper. *What every programmer should know about memory*. Technical report. Red Hat, Inc, 2007, pages 1–114.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th. Morgan Kaufmann, 2011.

[6] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[7] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. "Rabbit order: Just-in-time parallel reordering for fast graph analysis". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pages 22–31.

[8] J. Heaton. "Comparing dataset characteristics that favor the Apriori, Eclat or FP-Growth frequent itemset mining algorithms". In: *SoutheastCon 2016*. IEEE. 2016, pages 1–7.

[9] H. Wei, J. X. Yu, C. Lu, and X. Lin. "Speedup graph processing by graph ordering". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pages 1813–1828.

[10] T. Messi Nguélé, M. Tchuente, and J.-F. Méhaut. "Social network ordering based on communities to reduce cache misses". In: *Revue ARIMA* Volume 24 - 2016-2017 - Special issue CRI 2015 (May 2017).

[11] T. M. Nguélé, M. Tchuente, and J.-F. Méhaut. "Using Complex-Network Properties for Efficient Graph Analysis". In: *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*. 2017, pages 413–422.

[12] F. Gamgne Domgue, N. Tsopze, and R. Ndoundam. "UCAD: commUnity disCovery method in Attribute-based multicoloreD networks". In: *Knowledge and Information Systems* 66.10 (2024), pages 6337–6362. ISSN: 0219-3116.

[13] N. Merkel, P. Toussing, R. Mayer, and H.-A. Jacobsen. "Can graph reordering speed up graph neural network training? An experimental study". In: *Proceedings of the VLDB Endowment* 18.2 (2024), pages 293–307.

[14] R. A. Mogo Wafo, T. Messi Nguelé, A. J. Nzekon Nzeko'o, and D. Youh Xaviera. "Clustering-based Graph Numbering using Execution Traces for Cache Misses Reduction in Graph Analysis Applications". In: *Revue Africaine de Recherche en Informatique et Mathématiques Appliquées* 43 (Feb. 2025).

[15] R. Zhang, J.-A. Chen, H.-H. Sung, A. Li, and X. Shen. "N:M sparsity-oriented graph reordering for accelerating GNNs on GPU sparse tensor cores". In: *Journal of Parallel and Distributed Computing* 213 (2026), page 105264. ISSN: 0743-7315.

A ACKNOWLEDGEMENTS

The successful completion of this research has been made possible through the generous support and collaboration of various scholarship stakeholders. We express our sincere gratitude to the International Development Research

Centre (IDRC), the Swedish International Development Cooperation Agency (SIDA), and the African Center for Technology Studies (ACTS) for awarding the Artificial Intelligence for Development (AI4D) scholarship program that funded this research. This scholarship not only provided financial support but also served as a strong source of motivation and encouragement throughout the project.

We would also like to acknowledge the Agence Universitaire de la Francophonie (AUF) for its support at the early stage of this work. This support provided a favorable research environment and enabled the initial development of the ideas presented in this paper, notably through collaboration with *Cheikh Hamidou Kane Digital University*(UNCHK).

We are grateful for the support of all members of our work team, namely High performance data science (HIPERDAS, hosted in the ARITeD association), who contributed to stimulating discussions and shared their insights. The collaborative environment fostered by our academic community has been integral to the development of this work.

Lastly, we extend our thanks to all those who, directly or indirectly, played a role in the realization of this paper. Your support has been crucial, and we are grateful for the opportunities provided by the scholarship stakeholders and the broader academic community.

B BIOGRAPHY

Regis Audran Mogo Wafo is a Ph.D. student at the Department of Computer Science of the University of Yaoundé I. His research focuses on graph processing, memory access optimization, and performance improvement of graph analysis applications. His work particularly explores execution trace analysis, cache-efficient data layouts, and the use of clustering and frequent itemset mining techniques to enhance data locality and reduce execution time.

Thomas Messi Nguele received his Ph.D. degree from Grenoble Alpes University and the University of Yaoundé I in 2018. He is currently Head of the Department of Computer Engineering at the University of Ebolowa and a Senior Lecturer at the University of Yaoundé I. His research interests include parallel machine learning, machine learning, and recommender systems.

Armel Jacques Nzekon Nzeko'o received his Ph.D. degree from Sorbonne University and the University of Yaoundé I in 2019. He is currently a Senior Lecturer at the University of Yaoundé I. His research interests include recommender systems, machine learning, and parallel machine learning.

Xaveria Youh Djam received her Ph.D. degree from the University of Yaoundé I in 2014. She is currently a Professor and Senior Software Engineer at the University of Yaoundé I. Her research interests include software engineering, software testing, and artificial intelligence applications.