ARIMA

épisciences

Applying Data Structure Succinctness to Graph Numbering For Efficient Graph Analysis

Thomas MESSI NGUELÉ[*1] **and Jean-François MÉHAUT**[2]

[1]University of Yaounde 1, Cameroon
[2]University Grenoble Alpes , France

*E-mail : thomas.messi@facsciences-uy1.cm

DOI : 10.46298/arima.8349
Submitted on August 10, 2021 - Published on January 17, 2022
Volume : **Volume 32 - 2019 - 2021** - Year : **2022**
Special Issue : **Volume 32 - 2019 - 2021**
Editors : *Eric Badouel, Nabil Gmati, Maurice Tchuenté, Bruce Watson*

Abstract

Graph algorithms have inherent characteristics, including data-driven computations and poor locality. These characteristics expose graph algorithms to several challenges, because most well studied (parallel) abstractions and implementation are not suitable for them. In previous work [17, 18, 20], authors show how to use some complex-network properties, including community structure and heterogeneity of node degree, to improve performance, by a proper memory management (Cn-order for cache misses reduction) and an appropriate thread scheduling (comm-deg-scheduling to ensure load balancing). In recent work [19], Besta et al. proposed log(graph), a graph representation that outperforms existing graph compression algorithms. In this paper, we show that graph numbering heuristics and scheduling heuristics can be improved when they are combined with log(graph) data structure. Experiments were made on multi-core machines. For example, on one node of a multi-core machine (Troll from Grid'5000), we showed that when combining existing heuristic with graph compression, with Pagerank being executing on Live Journal dataset, we can reduce with cn-order: **cache-references** from ***29.94%*** (without compression) to ***39.56%*** (with compression), **cache-misses** from ***37.87%*** to ***51.90%*** and hence **time** from ***18.93%*** to ***28.66%***.

Keywords

graph ordering, cache misses reduction, graph compression

I INTRODUCTION

Many systems are modeled as a graph G = (V, E) where entities are represented by vertices in V, and connections are represented by edges in E. The advent of computers and communication networks allows to analyze data on a large scale and has lead to a shift from the study of

individual properties of nodes in small specific graphs with tens or hundreds of nodes, to the analysis of macroscopic and statistical properties of large graphs also called complex networks, consisting of millions and even billions of nodes [5].

This huge size makes the design of efficient algorithms for parallel complex network analysis a real challenge [6]. The first difficulty is that graph computations are data driven, with high data access to computation ratio. As a consequence, memory fetches must be managed very carefully, otherwise as observed in other domains such as on-line transactions processing, the processors can be stalled up to 50% of the time due to cache misses [3].

On the other hand, the irregular structure of complex networks combined with the spatial locality of graph exploration algorithms, make difficult the scheduling task. Indeed, the workload of the thread generated when dealing with the neighbors of a node u depends on the degree of u. As a consequence, because of the great heterogeneity of nodes degrees, these threads are highly unbalanced. In this regard, it has been shown that in some graph applications, execution time can be reduced by more than 25% with a proper scheduling which ensures that, threads that are executed together on a parallel computer have balanced load [7, 9].

Besta et al. proposed log(graph) [19], a graph representation that uses high compression ratios with very low-overhead decompression. They showed that they can ensure storage reduction of 20%-30% and still ensure graph processing acceleration. The aim of this paper is to use their compression approach together with our previously proposed heuristics (graph order and scheduling).

Contribution. In this paper, we combine log(graph) techniques [19] such as ID logarithmization (global approach and local approach) with:

- Graph ordering heuristics:Cn-order[18, 20], Gorder [16], Rabbit [15] and NumBaCo [17]). We expected compression will allow to increase cache references reduction, cache misses reduction and consequently execution time reduction.
- Scheduling heuristics [18, 20]: these heuristics combine graph ordering heuristics advantages with a scheduling that ensure load balancing among threads during graph applications. With compression, we expected to improve performances even in a context of multi-thread application.

Paper organization. The remainder of this paper is structured as follows. Section II presents prerequisites such as graph representation, cache management, Common pattern in graph analysis and logarithmization techniques. Section III presents recent works on graph ordering and graph compression. Section IV introduces our main contribution: the impact of compression in graph ordering performances. Experimental results are shown in section V. Section VI is devoted to the synthesis of our contribution, together with the conclusion and future work.

II BACKGROUND

2.1 Complex Network Representation

As already said, complex networks are modeled with graphs. There are three main ways to represent these graphs: matrix representation, Yale representation and adjacency list represen-

tation. This section details each of these representations. Figure 1 is an illustration example. It is an undirected weighted graph with 8 nodes.

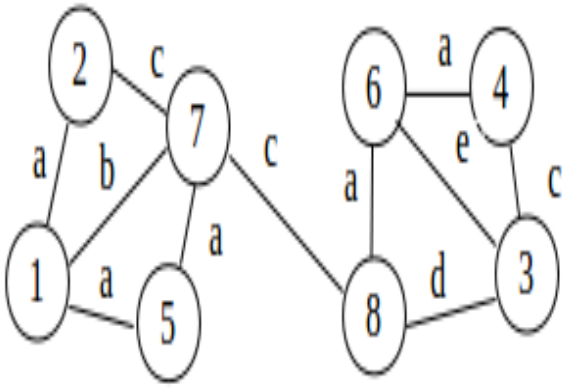

Figure 1: Example of graph (G1)

Matrix Representation. A simple way to represent graph is to use a matrix representation M where $M(i, j)$ is the weight of the edge between i and j. This is not suitable for complex graphs because the resulting matrices are very sparse. With the graph in figure 1, the total used space is 8x8 = 64. But 42 (i.e 66%) are wasted to save zeros.

Yale Representation. The representation often adopted by some graph specific languages such as Galois [13] and Green-Marl [11], is that of Yale [2]. This representation uses three vectors: A, JA and IA.

- A represents edges, each entry contains the weight of each existing edge (weights with same origin are consecutive),
- JA gives one extremity of each edge of A,
- IA gives the index in vector A of each node (index in A of first stored edge that have this node as origin).

Yale representation of the above example is in TABLE 1.

A	a	a	b	a	c	c	e	d	c	a	a	a	e	a	a	b	c	a	c	d	a	c
JA	2	5	7	1	7	4	6	8	3	6	1	7	3	4	8	1	2	5	8	3	6	7
IA	1	4	6	9	11	13	16	20	23													

Table 1: Yale representation of graph (G1)

Adjacency List Representations. Other platforms use adjacency list representations. In this case, the graph is represented by a vector of nodes, each node being connected either to a block of its neighbors [12] (see figure 2-a) or to a linked list of blocks (with fixed size) of its neighbors, adapted to the dynamic graphs, used by the Stinger platform in [8, 10] (see figure 2-b)

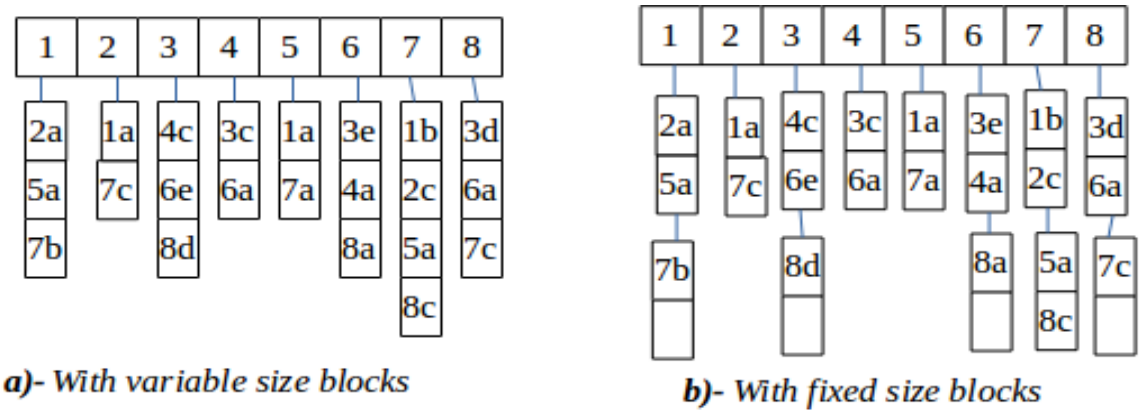

Figure 2: Adjacency list representations of (G1)

2.2 Cache Management

Many studies show that memory operations such as CPU cache latency and cache misses take most of the time on modern computers [3]. In this paragraph we show how cache memory is managed.

When a processor needs to access to data during the execution of a program, it first checks the corresponding entry in the cache. If the entry is found in the cache, there is cache hit and the data is read or written. If the entry is not found, there is a cache miss. There are three main categories of cache misses:

- Compulsory misses, caused by the first reference to data.
- Conflict misses, caused by data that have the same address in the cache (due to the mapping).
- Capacity misses, caused by the fact that all the data used by the program cannot fit in the cache.

Hereafter, we are interested in the last category.

In common processors, cache memory is managed automatically by the hardware (prefetching, data replacement). The only way for the user to improve memory locality, or to control and limit cache misses, is the way he organizes the data structure used by its programs. In this paper, we influence cache misses with numbering and compression. We do measures on a multi-core machine (presented at section V) with a famous tool, Linux Perf.

2.3 Log(graph) Formalization

In this section we present Log(graph) formalization as it was described in [19]. The authors call their approach "graph logarithmization" because they apply logarithmic terms to various graph elements: fine elements, offset structures, adjacency structures. In this paper we are interested to IDs logarithmization (global and local approach). We will illustrate it with Yale representation. For this representation, the space taken by a graph is:

$$\begin{cases} |A| = 2m * W_{edge} \\ |JA| = 2m * W_{node} \\ |IA| = n * W_{node} \end{cases}$$

Where n is the number of nodes, m the number of edges, W_{edge} the size in memory of an edge and W_{node} the size in memory of a node.

Vertex IDs logarithmization with global approach. In many designs, a vertex ID uses $W_{node} \in \{32, 64\}$ bits of space. That corresponds to the size of 32- or 64-bits memory word. Log(Graph) extends this approach and uses the lowest applicable value which is $W = \lceil log(n) \rceil$ bits. With this value, we have:

$$\begin{cases} |JA| = 2m * \lceil log(n) \rceil \\ |IA| = n * \lceil log(n) \rceil \end{cases}$$

Vertex IDs logarithmization with local approach. Even if previous approach uses an optimal number of bits to store n vertex IDs, it is not optimal when considering subsets of these vertices; especially for nodes that a very small ID compared to n. To overcome this situation, the authors use $\lceil log(N_v) \rceil$ bits to store a vertex ID and its neighbors (where N_v is v neighbor with the maximum ID). The trade off is that the information about the number of bits used should be kept for every node. So the space taken by this approach is:

$$\begin{cases} |JA| = \sum_{v \in V} (d_v \lceil log(N_v) \rceil + \lceil loglog(N_v) \rceil) \\ |IA| = \sum_{v \in V} (\lceil log(v) \rceil + \lceil loglog(v) \rceil) \end{cases}$$

III RELATED WORK

For several years, graph reordering algorithms have attracted the attention of researchers and this attention is constantly increasing. In our previous research, we compared three graph reordering that were design quite a the same period: NumBaCo [17], Rabbit Order [15] and Gorder [16]. Each of them pretends to outperform the existing graph reordering designed before like Slashburn [14]. This comparison results to the design of Cn-order and Com-deg-scheduling [18, 20] that showed better performances compared to the first three. Since this period, a lot of work was done around graph ordering.

Reet Barik et al. [21] study some vertex reordering algorithm with two graph applications: community detection and influence maximization. Their study doesn't show clearly the impact of cache references reduction, cache misses reduction and degree aware scheduling in the resulting performance induced by those ordering as it is done when presenting Cn-order in[18, 20].

Mohsen Koohi et al. [23] study three reordering algorithms such as Slashburn [14], Gorder [16] and Rabbit order [15]. Their study suggests that memory locality improvement is related to the distribution of node degrees of the target graph. The author did not include Cn-order in their study. They didn't clearly identify the best graph reordering algorithm and didn't explain why it is the best.

Benjamin Coleman et al. [22] use some graph reordering algorithms in order to reduce cache misses while doing near neighbor searching. The author didn't include Rabbit order and Cn-order that are suppose to be some of the best reordering.

None of the recent works presented in this section doesn't include Cn-order [18, 20]. In this paper, we extend our previous work done on Cn-order and show that performances can be

improved when combining graph ordering techniques with IDs logarithmization approach proposed by Besta et al. [19].

IV IMPACT OF GRAPH COMPRESSION IN GRAPH ORDERING

In this section, in order to present the impact of graph compression in Graph Ordering Problem, we first recall the common pattern in graph analysis and give an illustration example. After that we recall the cache model, we show how one can estimate the number of cache misses with and without compression, then we present the problem of graph ordering for cache misses reduction and its complexity. And finally we present two algorithms: **ComNumGra** that combines compression and graph ordering, and **ComNumGra-scheduling** that combines the later with degree-aware scheduling [18, 20].

4.1 Common Pattern in Graph Analysis

One common statement used in graph analysis is as follows:

1: **for** $u \in V(G)$ **do**
2: **for** $v \in Neighbor(u)$ **do**
3: program section to compute/access v
4: **end for**
5: **end for**

With this pattern, one should pay attention both in accessing u and v. In order to improve performances (with cache misses reducing), one should ensure that successive u and v are close in memory.

4.2 Illustration Example

Figure 3 presents the example used in this section. Suppose an analysis algorithm executes the previous pattern for the first two nodes, 0 with $Neig(0) = \{4, 11\}$ and 1 with $Neig(1) = \{5, 10, 13\}$. The accessing sequence is: $N_6 = (\mathbf{0}, 4, 11, \mathbf{1}, 5, 10, 13)$. With the data cache provide in figure 4, this will cause 7 cache misses (each access of a node will cause a cache miss).

But if we consider one of the orders of the same graph, Rabbit order for example, the number of cache misses is reduced because consecutive nodes are closed in memory.

4.3 Cache Modeling

We consider capacity cache misses caused by the fact that data used by a program cannot fit in the cache memory. This cache model also considers a memory cache with one line and t blocks in main memory (see figure 4).

When a node x is being processed (x is in cache memory), two situations are envisaged while trying to access another node y:

- if x and y are in the same memory block $b(x) = b(y)$, then y is also in cache memory. $b(x)$ gives the belonging block of a node x in memory.
- If x and y are not in the same memory block, there is a cache miss.

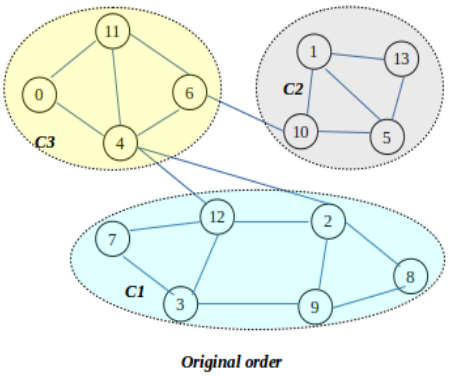

Figure 3: Running example

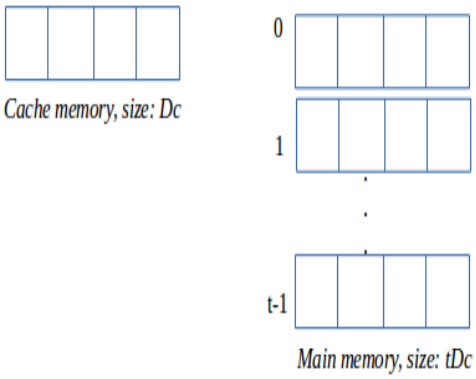

Figure 4: Memory representation

A cache miss can be modeled by σ as follows:

$$\begin{array}{rccl} \sigma: & N \times N & \rightarrow & \{0,1\} \\ & (x,y) & \mapsto & \sigma(x,y) = \begin{cases} 0 & if\ b(x) - b(y) = 0 \\ 1 & else \end{cases} \end{array}$$

4.4 Number of Cache Misses in Program

Let P be a program on a graph $G = (N, E)$. P makes reference to an ordered sequence of nodes noted by $< P >$. We assume that this sequence of nodes is scattered throughout l memory blocks, each block B_i containing l_i nodes of that sequence noted by $< B_i >$.

That is: $\begin{cases} < P >= (n_1, n_2, n_3, ..., n_k) =< B_1 >< B_2 > ... < B_L > \\ \text{With } < B_i >= (n_{i_1}, n_{i_2}, n_{i_3}, ..., n_{i_{l_i}}) \end{cases}$

The number of cache misses caused by the execution of P can be estimated by:

$$CacheMiss_G(< P >) \begin{cases} = 1 + \sum_{i=2}^{k} \sigma(n_i, n_{i-1}) \\ = \sum_{i=1}^{l} CacheMiss(< B_i >) \\ = \sum_{i=1}^{l} \sum_{j=2}^{l_i} (1 + \sigma(n_{i_j}, n_{i_j - 1})) \end{cases} \quad (1)$$

4.5 Graph Compression Effects in Cache Misses

We consider a compression scheme that reduces the size of each node of the graph in memory. A block memory containing s nodes without compression may contain $s' > s$ after compression. Let $1/\beta$ be the compression ratio ($\beta \in]0, 1[$). The number of nodes that may be added in each block is:

$$s' - s = s(1/\beta - 1) \quad (2)$$

In equation 1, we estimate the number of cache misses as the sum of the number of cache misses of each block B_i. In the same way, we can consider the number of cache misses reduced due

to compression as the sum of cache misses reduced in each block. The maximum number of cache misses reduced per block corresponds to the number of added nodes per block.

Finally, due to compression, we expect to reduce at most $ls(1/\beta-1)$ cache misses. Considering that limit as the number of cache misses reduced, we have:

$$CacheMiss(< P >) = \sum_{i=1}^{l} \sum_{j=2}^{l_i} (1 + \sigma(n_{i_j}, n_{i_j-1})) - ls(1/\beta - 1) \tag{3}$$

4.6 Numbering Graph Problem

We are looking for a permutation π of G's nodes in such a way that the execution of P produces the minimum number of cache misses (min). In other words:

Let $G = (N, E)$ and $min \in \mathbb{N}$,

$$\begin{cases} - & Find \quad \pi : \mathbb{N} \to \mathbb{N} \\ & \qquad\quad n \mapsto \pi(n) \\ - & such\ that \ \sum_{i=1}^{l} \sum_{j=2}^{l_i} [1 + \sigma(\pi(n_{i_j}), \pi(n_{i_j-1}))] - ls(1/\beta - 1) \leq min, \\ - & with \quad < P >=< B_1 B_2 ... B_L >, B_i = (n_{i_1}, n_{i_2}, n_{i_3}, ..., n_{i_{l_i}}) \ and \ n_{i_j} \in N. \end{cases}$$

***Theorem** IV.1: complexity of the problem*
The Numbering Graph Problem (NGP) to minimize the number of cache misses is NP-complete.

Proof:. This proof can be done with the Optimal Linear Arrangement Problem (OLAP) known as NP-Complete [1]. As reminder, this problem is defined as follows:

$$Let\ G = (N, A), min \in \mathbb{N} \begin{cases} - & Find \quad \pi : \mathbb{N} \to \mathbb{N} \\ & \qquad\quad n \mapsto \pi(n) \\ - & such\ that \ \sum_{\{n_i, n_j\} \in A} |\pi(n_i) - \pi(n_j)| \leq min \end{cases}$$

To show that NGP is NP-complete, we have to show that any instance of OLAP is polynomial time reduced to NGP. In that way, it is easy to remark that any instance of OLAP is an instance of NGP when considering the execution of the loop which traverses all the edges of G.

4.7 Graph ordering and graph compression

In this section, we propose a simple heuristic that solves the numbering graph problem for cache misses reduction. This solution is described in algorithm 1 (**ComNumGra**). This algorithm has two steps:

- The numbering step that consists of executing any ordering heuristics such as Cn-order [18, 20], NumBaCo [17], Rabbit [15] or Gorder [16].
- The compressing step. In this step, we consider IDs logarithmization approach from [19]. We choose it because the decompression has almost no cost.

Algorithm 1 : **ComNumGra** (**Com**press and **Num**ber **Gra**ph)

Input: $G = (V, E)$
Output: $G'' = \sigma(\pi(G))$, π permutation, σ is a graph compressor

1: $G' \leftarrow numbering\_graph(G)$; *//Cn-order for example*
2: $G'' \leftarrow compressing\_graph(G')$; *//log(graph) for example*;
3: Return G''

4.8 Compression and Scheduling

We showed in [18, 20] that **deg-scheduling** algorithm keeps nodes consecutive for each thread which contributes to increase performance when using numbering algorithms such as Cn-order. We claim that combining deg-scheduling with ComNumGra will also probably increase performance. This is done in ComNumGra-scheduling (algorithm 2).

Algorithm 2 :ComNumGra-scheduling

Input: $G = (N, E)$,d_{sum}, nb_{thread}
Output: $task[\,]$, a vector where each task has start and end nodes

1: $G'' \leftarrow ComNumGra(G)$;
2: $task[\,] \leftarrow$deg-scheduling($G'', d_{sum}, nb_{thread}$);

In the next section, we will show experimentally how these algorithms improve previous results in terms of cache reference reduction, cache misses reduction, and hence execution time.

V EXPERIMENTAL EVALUATION

The experiments were done on one node of a multi-core machine (Troll from Grid'5000). This machine has the following characteristics: 4 nodes, each node has 2 CPUs Intel Xeon Gold 5218, each CPU has 16 cores, 384GB RAM, L3 of 22 MB, L2 of 1024 KB, L1 of KB.

For this evaluation, we present results got with the well known graph analysis application, Pagerank [4]. We used a posix thread implementation proposed by *Nikos Katirtzis*[1]. This implementation uses adjacency list representation. A node in a graph is represented by six fields:

- two are int ($2 * sizeof(int)$): these fields contains the number of in-neighbors and the the number of out-neighbors respectively;
- three fields are double ($3 * sizeof(double)$): these fields contain other pagerank information used during the computation;
- one is a pointer to int ($sizeof(int*)$): this field may contain the out-neighbors of the current node.

[1] https://github.com/nikos912000/parallel-pagerank

Each node may take $S = 2 * sizeof(int) + 3 * sizeof(double) + sizeof(int*)$ (without counting the neighbors space). With a graph of N nodes and m edges, that will take $N * S + \{neighbor\ space\}$. Since, we consider non oriented graphs, the total space taken by neighbors is $2m * sizeof(int)$. Finally, the total space of the graph is *G_space* given by:

$$\begin{cases} G\_space & = N * S + 2m * sizeof(int) \\ & = 2m * sizeof(int) + N * S \\ & = 2m * sizeof(int) + N * (2 * sizeof(int) + 3 * sizeof(double) + sizeof(int*)) \end{cases}$$

In the used architecture, an int uses four bytes, a $float$ uses eight bytes, an $int*$ uses eight bytes. In that way, $G\_space = 2m * 4 + N * (2 * 4 + 3 * 8 + 8) bytes = 8m + 40N\ bytes$:

- With N= 3,997,962 nodes and m= 34,681,189 edges, the approximate space taken by Live Journal in memory is: 8* 34,681,189 + 40 * 3,997,962 = 417.10*2^{20} bytes = **417.10 MB**.
- with N= 3,072,441 nodes and m=117,185,083 edges, the approximate space taken by Orkut is: 8*117,185,083+40*3,072,441= 1011.25*2^{20} bytes =**1011.25 MB**.

We present the effects of compression on graph ordering in two ways: on a single core (one thread) and on multi-core (32 threads).

5.1 Effect of Compression on Graph Ordering On a Single core

Results in tables 2 and 3 confirm the strengths and weaknesses of each graph ordering already presented in [17, 18, 20]. These results show that the compressing improves the previous observed performances and almost in the same way.

5.1.1 Compression effects on Live Journal data set

Table 2 presents the effect of compression in pagerank with Live Journal data set. On this table, we can see that:

- CN-order and NumBaCo outperform the others graph ordering in terms of:
  - $Nodes\_closeness$. NumBaCo is with 53.61%
  - Cache-references. CN-order with 29.94%
  - Cache-misses. NumBaCo is the best by with 38.24%
  - Time. CN-order reduces the original time by 18.93%
- Compressing the graph with id logarithmization in its global approach (Press1) improves the previous results in the same trend. That is, CN-order and NumBaCo outperforms when using compression:
  - $Nodes\_closeness$. NumBaCo is still the best with 53.61% (compression has no effect on $node\_closeness$).
  - Cache-references. CN-order is the best with 39.56% (compared to 29.94%)
  - Cache-misses. NumBaCo is the best with 51.98% (compared to 38.24%)
  - Time. CN-order reduces the original time by 28.66% (compared to 18.93%).
- Ids logarithmization in its local approach produces very poor results.

In section 5.1.3, we explain the reason why the compression with Ids logarithmization with global approach allow us to improve performances but not the Ids logarithmization with local approach.

Table 2: Compression effects (Pagerank, Live Journal, 1 thread, troll)

Heuristic	Time (ms)	Cache miss (millions)	Cache ref (millions)	*nodes_closeness*
Original	54944.5	3643.76	6208.32	38,631
Press1(Original)	57603.2	3148.74	5801.77	
Press2(Original)	83444.4	7021.03	9585.08	
Gorder	51403.39	3252.59	4915.97	41,407
	(6.44%)	(10.73%)	(20.82%)	(+7.19%)
Press1(Gorder)	51387.81	2668.31	4348.81	
	(6.47%)	(26.77%)	(29.95%)	
Press2(Gorder)	73689.85	6399.28	8012.60	
NumBaCo	45346.35	2250.23	4724.24	17,922
	(17.47%)	**(38.24%)**	**(23.90%)**	**(53.61%)**
Press1(**NumBaCo**)	39859.14	1749.62	4130.14	
	(27.46%)	**(51.98%)**	**(33.47%)**	
Press2(**NumBaCo**)	66339.09	5564.48	8025.66	
Rabbit	48840.08	2517.65	4831.27	26,390
	(11.11%)	(30.90%)	(22.18%)	(31.68%)
Press1(Rabbit)	44393.94	2043.62	4289.98	
	(19.20%)	(43.91%)	(30.90%)	
Press2(Rabbit)	70339.86	5840.22	8108.50	
cn-order	44542.016	2263.81	4349.19	19,734
	(18.93%)	**(37.87%)**	**(29.94%)**	**(48.92%)**
Press1(**cn-order**)	39194.96	1752.61	3752.28	
	(28.66%)	**(51.90%)**	**(39.56%)**	
Press2(**cn-order**)	64715.40	5554.28	7566.18	

5.1.2 Compression effects on Orkut data set

Table 3 presents the effect of compression in pagerank with Orkut data set. On this table, we can see that:

- If CN-order and NumBaCo outperform the others graph ordering in terms of:
  - $Nodes\_closeness$. NumBaCo is with 70.46%
  - Cache-misses. CN-order is the best with 43.05%
  - Time. CN-order reduces the original time by 30.96%
- Gorder is the best in reducing cache references by 28.44%.
- As in table 2, compressing the graph with id logarithmization in its global approach (Press1) improves the previous results in the same trend. That is:
  - $Nodes\_closeness$. NumBaCo is still the best with 70.46% (compression has no effect on $node\_closeness$).
  - Cache-references. Gorder is the best with 33.27% (compared to 28.44%)
  - Cache-misses. CN-order is the best with 55.13% (compared to 43.05%)
  - Time. CN-order reduces the original time by 35.02% (compared to 30.96%).
- Ids logarithmization in its local approach produces very poor results.

5.1.3 Explaining the compression effect

We have already explained in [17, 18, 20] that the reason why CN-order is globally the best is that it combines the strengths of two strategies situated at extreme positions: Numbaco strategy (best in cache misses reduction) and Gorder strategy (best in cache references reduction). So, CN-order tries to reduce both cache misses and cache references and finally it reduces the execution time. In this section, we explain the effect of compression in performances (cache references, cache misses and time).

Table 3: Performances comparison (Pagerank, Orkut, 1 thread, troll)

Heuristic	Time (ms)	Cache miss (millions)	Cache ref (millions)	*nodes_closeness*
Original	130811.35	8266.25	17015.58	148,080
Press1(Original)	138531.54	7254.20	16417.32	
Press2(Original)	211285.176	16911.85	25425.15	
Gorder	106101.93	6965.66	12175.18	151,204
	(18.89%)	(15.73%)	**(28.44%)**	(+2.11%)
Press1(Gorder)	110787.66	5915.41	11354.76	
	(15.31%)	(28.44%)	**(33.27%)**	
Press2(Gorder)	174866.26	15193.70	20331.73	
NumBaCo	96135.732	4712.83	14528.25	43,735
	(26.50)%	**(42.99%)**	(14.62%)	**(70.46%)**
Press1(NumBaCo)	88789.07	3753.84	13711.65	
	(32.12%)	**(54.58%)**	(19.41%)	
Press2(NumBaCo)	164688.12	13010.76	23109.20	
Rabbit	99136.22	4773.08	15129.13	89,596
	(24.21%)	(42.26%)	(11.09%)	(39.49%)
Press1(Rabbit)	90998.53	3804.27	14263.47	
	(30.43%)	(53.97%)	(16.17%)	
Press2(Rabbit)	167639.23	13017.03	23729.08	
cn-order	90313.87	4707.42	11814.20	50,448
	(30.96%)	**(43.05%)**	**(30.57%)**	**(65.93%)**
Press1(cn-order)	84997.19	3709.19	10916.39	
	(35.02%)	**(55.13%)**	**(35.84%)**	
Press2(cn-order)	153958.84	12854.18	20073.75	

Global approach of IDs logarithmization. Let max_degree be the maximum degree of a node, that is maximum number of neighbors a node may have. Using the global approach of ids logaritmization suggests that:

- The fields representing the number of in-neighbors and the number of out-neighbors should take $\lceil log_2(max\_degree) \rceil$ Bytes.
  - With Live Journal graph, the max degree is $14815 \in]2^8, 2^{16}[$. So 2 bytes are needed instead of 4 bytes used before.
  - With Orkut graph, the max degree is $333135 \in]2^8, 2^{16}[$. So 2 bytes are needed instead of 4 bytes used before.
- Each node in the neighbor array should take $\lceil log_2(N) \rceil$ Bytes.
  - With Live Journal graph, N is $3,997,962 \in]2^{16}, 2^{24}[$. So 3 bytes are needed instead of 4 bytes used before.
  - With Orkut graph, the max degree is $3,072,441 \in]2^{16}, 2^{24}[$. So 3 bytes are needed instead of 4 bytes used before.

Finally, with those specifications:

$$\begin{cases} G\_space_g & = 6m + 36N \ bytes \\ G\_space_{LJ} & = (6 * 34681189 + 36 * 3997962) = \textbf{335.70 MB, (19.51\%)} \\ G\_space_{orkut} & = (6 * 117185083 + 36 * 3072441) = \textbf{776.02 MB, (23.26\%)} \end{cases}$$

We can see that, compared to the results got in [17, 18, 20], the reduction of 19.51% (Live Journal) or 23.26% (Orkut) of the size of the graph (thanks to ids logarithmization local approach) is responsible of the performance increasing discussed in 5.1.1 and 5.1.2. The explanation behind is that:

- There is a higher concentration of nodes in pieces of graph. That is, each portion of the graph now contains 19.51% or 23.23% more nodes; that contributes to reduce cache references, cache misses and hence execution time.

- There is almost no cost for decompression. There is no extra cost for decompression with IDs logarithmization in its local approach.

Local approach of IDs logarithmization. In logarithmization with local approach, we add another field that takes one byte and is used to know the size of the maximum id ($maxID_v$) among the neighbors of a node v. The idea is that in the array representing the neighbors, instead of taking three bytes as the size of a neighbor like in the global approach, we may use one, two or three bytes (depending of $maxID_v$ size). We assume that the average is two bytes.

Finally, compared to the global approach, each node may take $S' = 37\ bytes$ (without counting the neighbors space). And each neighbor takes in average 2 bytes (instead of 3 bytes). This results to:

$$\begin{cases} G\_space & = N * S' + 2m * sizeof(neighbor) \\ & = 2m * (2\ bytes) + 37N\ bytes \\ & = 4m + 37N\ bytes \\ G\_space_{LJ} & = (4 * 34681189 + 37 * 3997962) = \textbf{273.37 MB, (34.46\%)} \\ G\_space_{orkut} & = (4 * 117185083 + 37 * 3072441) = \textbf{555.43 MB, (45.07\%)} \end{cases}$$

The compression ratio suggests that the results with local approach should be better than the one with global approach. Tables 2 and 3 show us the contrary. The main explanation is in the way a neighbor was represented: each neighbor was represented as a pointer to char that can an array of length one, two or three (depending of $maxID_v$. Unfortunately, a pointer two a char takes 8 bytes in the used architecture. This increase the space taken by the graph and then justifies the poor result. In other to avoid this result, one should use a strategy that does not represent neighbor with a pointer to char.

$$\begin{cases} G\_space & = N * S' + 2m * sizeof(neighbor) \\ & = 2m * (8\ bytes) + 37N\ bytes \\ & = 16m + 37N\ bytes \\ G\_space_{LJ} & = (16 * 34681189 + 37 * 3997962) = \textbf{670.26 MB, (+60.69\%)} \\ G\_space_{orkut} & = (16 * 117185083 + 37 * 3072441) = \textbf{555.43 MB, (+46.67\%)} \end{cases}$$

5.2 Effect of Compression on Graph Ordering with many threads

In section 5.1, we show that, with one core (one thread), compression improves the performances got with graph orders (Gorder, Rabbit, NumBaCo and Cn-order). In this section, we are studying the way this compression affects these graph orders with many cores (many threads).

In a multi-thread application, performances are not only influenced by cache misses reduction and cache references reduction, one should also consider load balancing among threads.

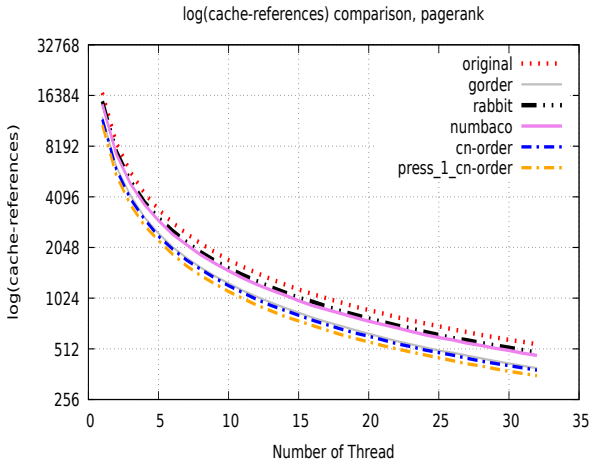

Figure 5: cache-ref with graph orders – Orkut

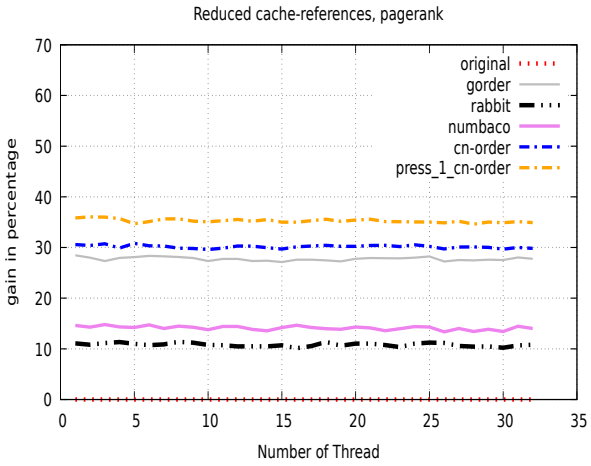

Figure 6: cache-ref with graph orders – Orkut

5.2.1 Effect of compression on Cache References Reduction

Figures 5, 6, 7, 8 (reported with Orkut dataset) and figures 9, 10, 11, 12 (reported with Live Journal dataset) represent the cache references gotten with pagerank. At the left of every figure, we have log of mean number of cache references per thread (from 1 to 32 threads). At the right, we have the gain in percentage (per number of thread) compared to the original (without using any heuristic). At each figure, we add another curve that represents the cache references gotten when using compression (press 1).

Orkut Dataset. With graph ordering heuristics in figures 5 and 6, we have observed as in our previous work that: Cn-order is the best with almost 30% of reduced cache references. It is close to Goder with almost 29%. NumBaCo is the third with almost 13% and Rabbit is the last with almost 10%. The gained percentage is nearly the same from one thread to 32 threads. The new observation here is that the compress version of the graph (thanks to logarithmization, global approach) allows to reduced references by a percentage more than 36% (from one to 32 thread). We have quite the same observations with .

We have quite the same observations with degree-aware scheduling heuristics in figures 7 and 8, Comm-deg-scheduling is the best with 30% of reduced cache references. But using compression improve cache references reduction to more than 36%.

Live Journal Dataset. In figures 9 and 10, with graph ordering heuristics, Cn-order reduces cache references by almost 30%. Compressing the graph contributes to reduce by around 40%.

We have the same observations with degree-aware scheduling heuristics in figures 11 and 12, where Comm-deg-scheduling is the best with 30% but this result is outperformed with compression.

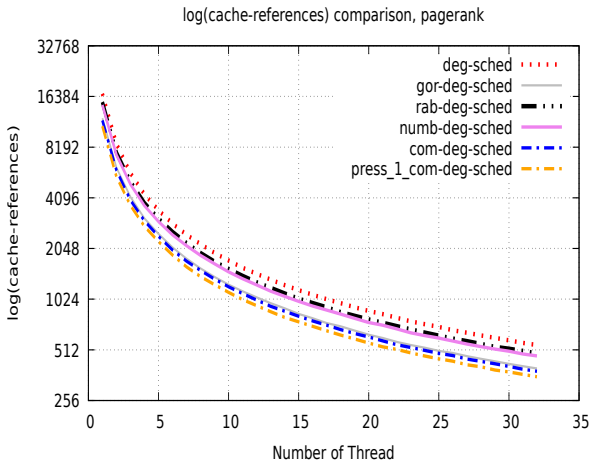

Figure 7: cache-ref with scheduling – Orkut

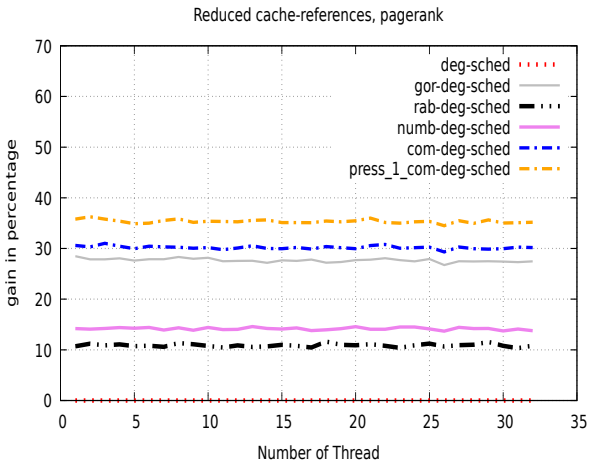

Figure 8: cache-ref with scheduling – Orkut

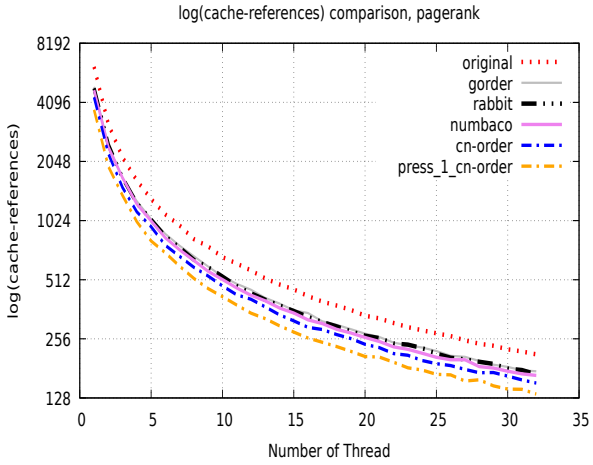

Figure 9: cache-ref with graph orders – Lj

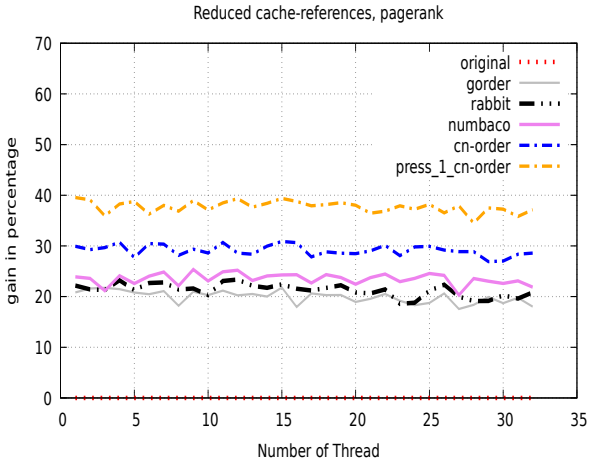

Figure 10: cache-ref with graph orders – Lj

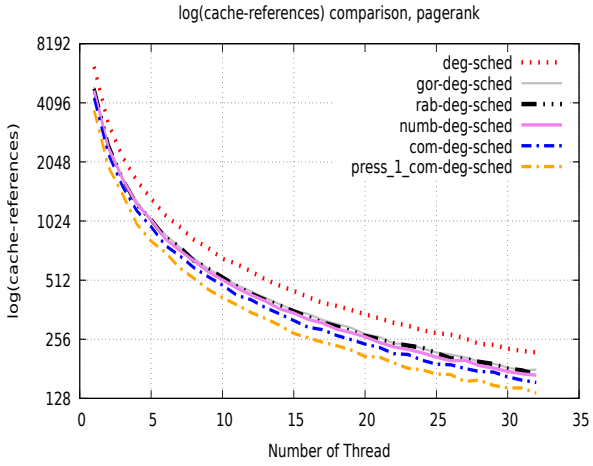

Figure 11: cache-ref with scheduling – Lj

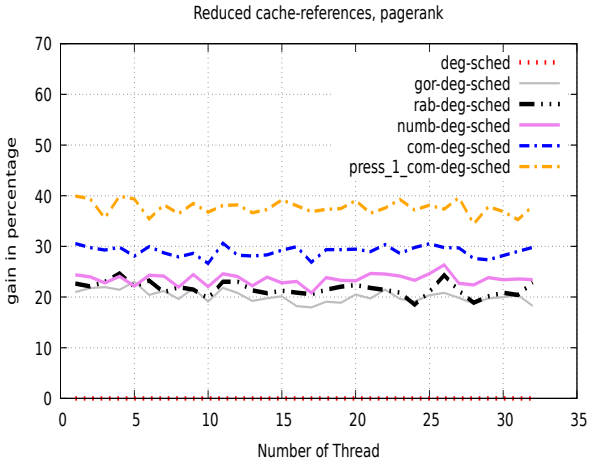

Figure 12: cache-ref with scheduling – Lj

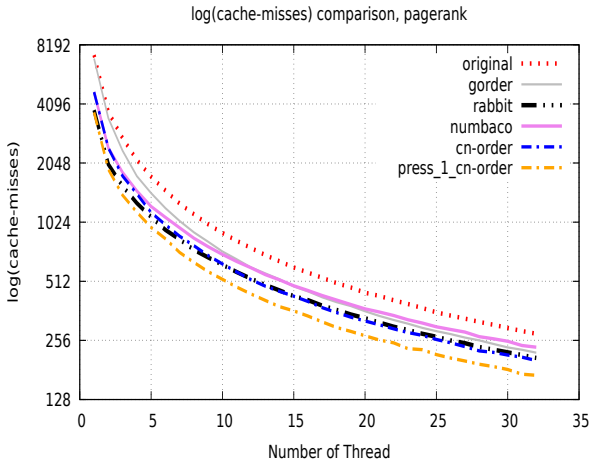

Figure 13: cache-misses with graph orders – Orkut

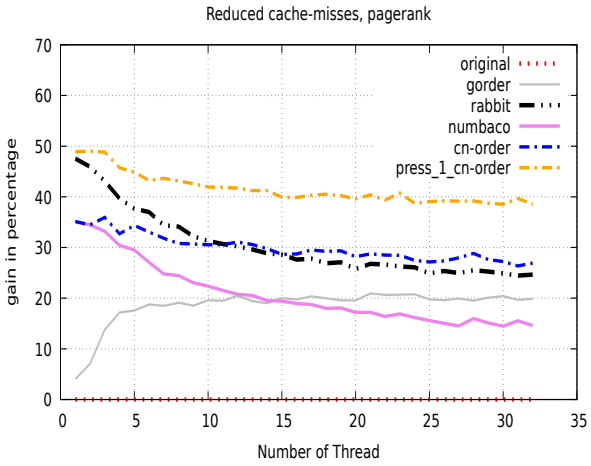

Figure 14: cache-misses with graph orders – Orkut

Interpretation. Considering the observations made with Orkut and Live Journal datasets on cache references reduction, we are able to say that:

- As already explained in our previous work, Cn-order is the best even with many threads (compared to Gorder, NumBaCo and Rabbit).
- Compressing the graph allow to improve the percentage of the reduction.

Does the compression improve the reduction of cache misses as it did with cache compression when using graph ordering heuristics and degree-aware scheduling heuristics? In the next section, we study cache misses reduction on the same datasets with Pagerank and will provide an answer to this question.

5.2.2 Effect of compression on Cache Misses Reduction

In this section, we discuss about cache misses reported with Orkut dataset present in figures 13, 14, 15, 16 and with Live Journal dataset in figures 17, 18, 19, 20. As with cache references reduction (section 5.2.1, at the left of every figure, we have log of mean number of cache misses per thread (from 1 to 32 threads) and at the right, we have the gain in percentage (per number of thread) compared to the original (without using any heuristic). We add another curve that represents the cache misses gotten when combining heuristic with compression.

Orkut Dataset. In figures 13 and 14, we can see that, with Cn-order, we have a reduction between 28% and 35% of cache misses, using compression allows to have a reduction between 40% and 49% of cache misses.

In figures 15 and 16, degree-aware scheduling heuristics have the same behavior as their homologous of graph ordering heuristics. Comm-deg-scheduling is the best with 32% - 42% of reduced cache misses. Using compression allows to reduce cache misses from 43% to 54%.

Live Journal Dataset. In figures 17 and 18, Cn-order and NumBaCo have the best cache misses reduction with almost 30% to 38%. When using compression, cache misses are now reduced from 45% to 52%.

In figures 19 and 20, as for Orkut dataset, degree-aware scheduling heuristics have also the same behavior as their homologous of graph ordering heuristics in cache misses reduction. And the using of compression increases the performances.

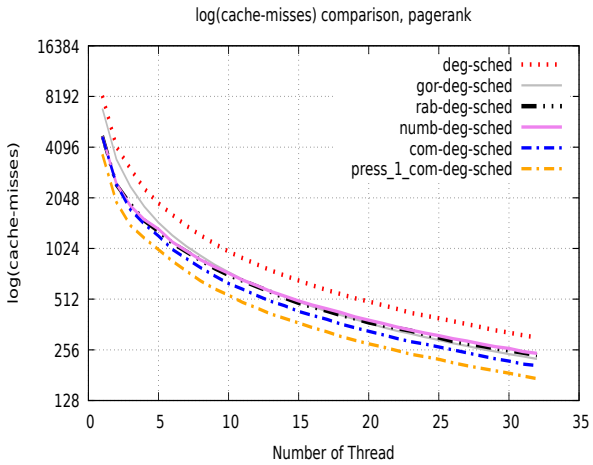

Figure 15: cache-misses with scheduling – Orkut

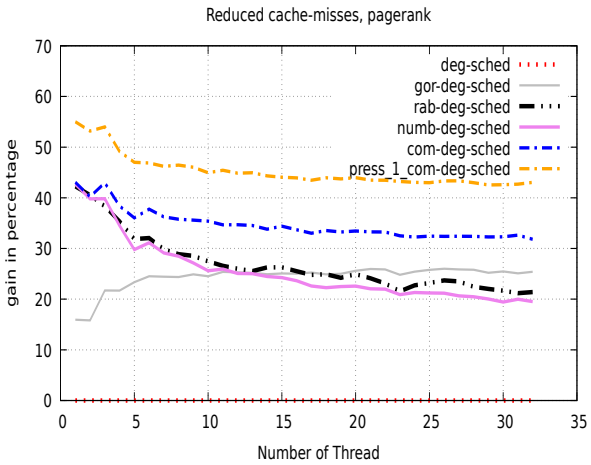

Figure 16: cache-misses with scheduling – Orkut

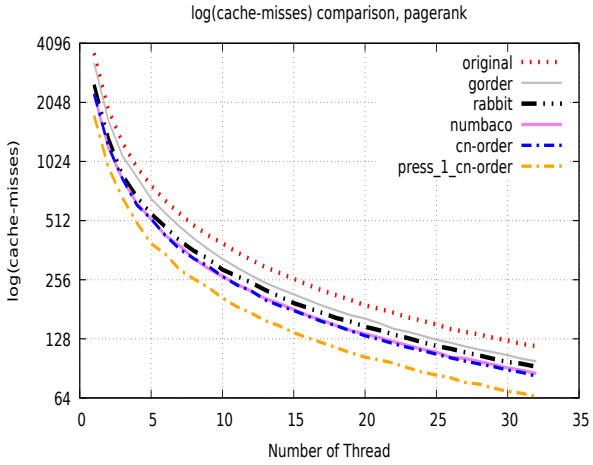

Figure 17: cache-misses with graph orders – Lj

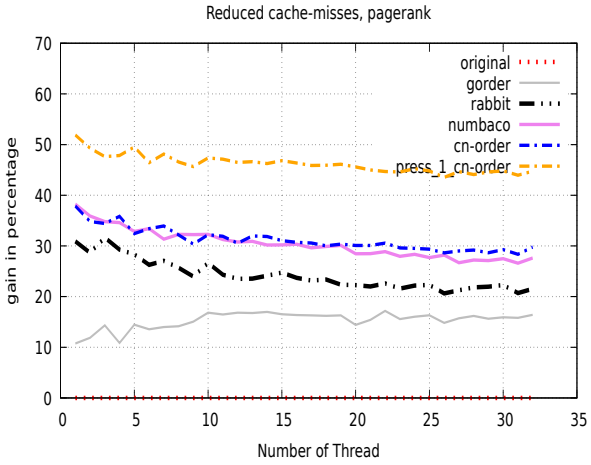

Figure 18: cache-misses with graph orders – Lj

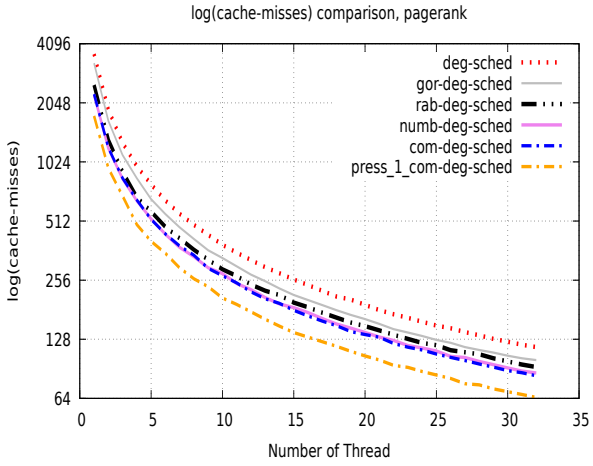

Figure 19: cache-misses with scheduling – Lj

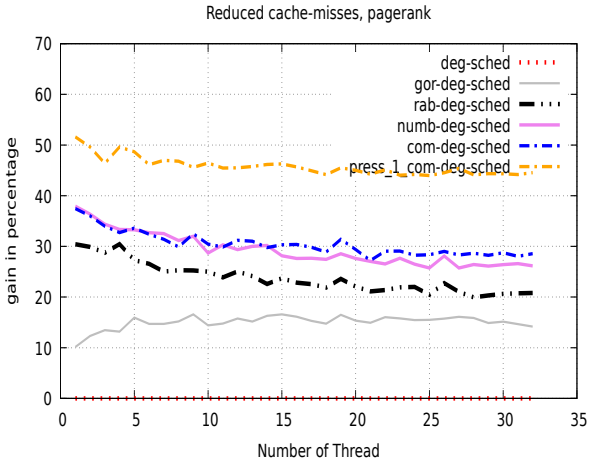

Figure 20: cache-misses with scheduling – Lj

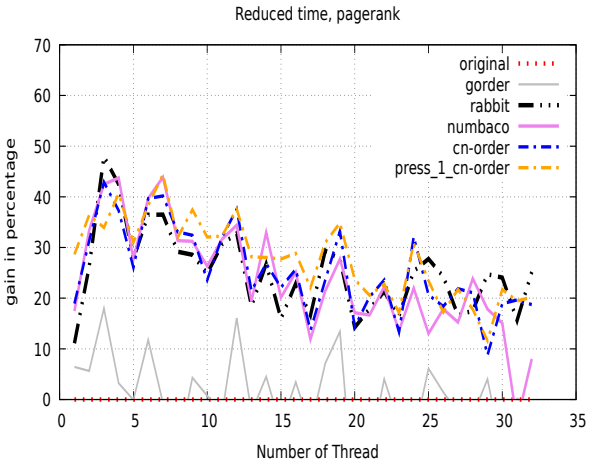

Figure 21: time with graph orders – Lj

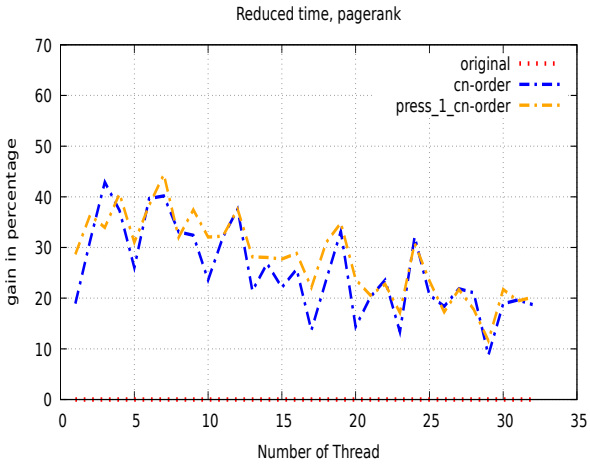

Figure 22: time reduced with graph orders – Lj

Interpretation. According to all the observations, we can say that:

- Combining graph order with compression improves cache misses reduction
- degree-aware scheduling heuristics with compression improves cache misses reduction.

In the next section, we discuss about the consequences of cache misses reduction and cache references reduction.

5.3 Impact in time reducing

Figures 21, 22, 23, 24 (gotten with Live Journal dataset) and figures 25, 26, 27, 28 (gotten with Orkut dataset) represent time reduction due to graph ordering heuristics (figures 21, 22, 25, 26) and degree-aware scheduling heuristics (figures 23, 24, 27, 28). We add at each figure a curve that shows the reduced time gotten with compression.

Live Journal Dataset. Remember that with one thread, Cn-order was the best in time reduction (compared to the other graph ordering heuristics). And even the compression contribute to reduce the time. In figures 21 and 22, we can see that, the curve gotten due to compression (press_1_cn-order is almost always on the top.

This position is more clear when using scheduling heuristics at figures 23 and 24. In both cases, compression produces positive effects in the performances.

Orkut Dataset. In figures 25, 26, 27 and 28, we have almost the same observation we had in the case of Live Journal dataset.

We showed in this section that combining graph compression with graph ordering heuristics and scheduling heuristics allows to improve the reduction of execution time; this is because it is the consequence of improving cache references reduction and cache misses reduction. In the next section, we conclude the paper.

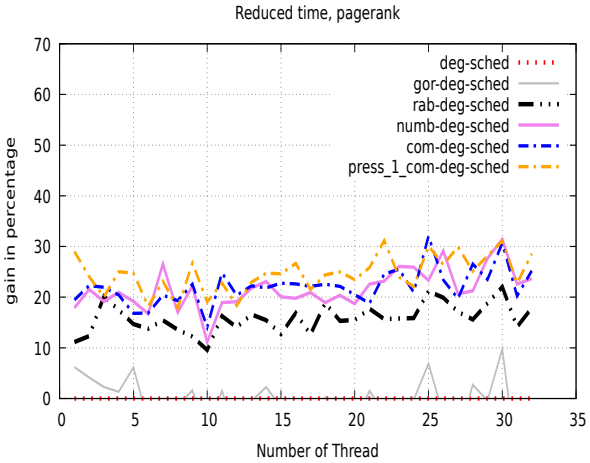

Figure 23: time with scheduling – Lj

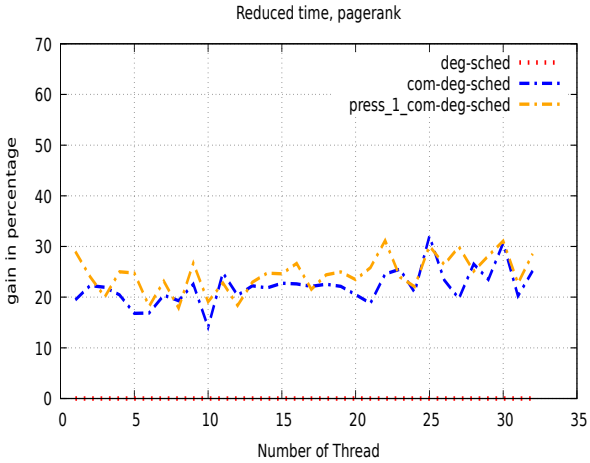

Figure 24: time reduced with scheduling – Lj

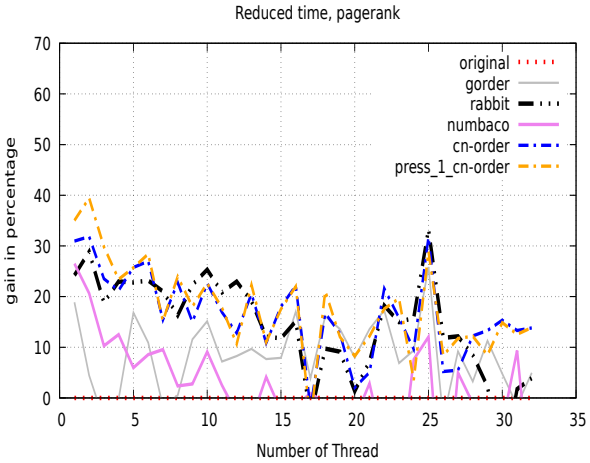

Figure 25: time with graph orders – Orkut

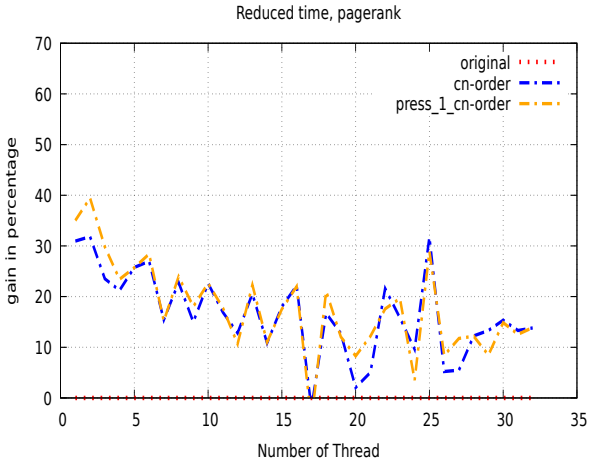

Figure 26: time reduced with graph orders – Orkut

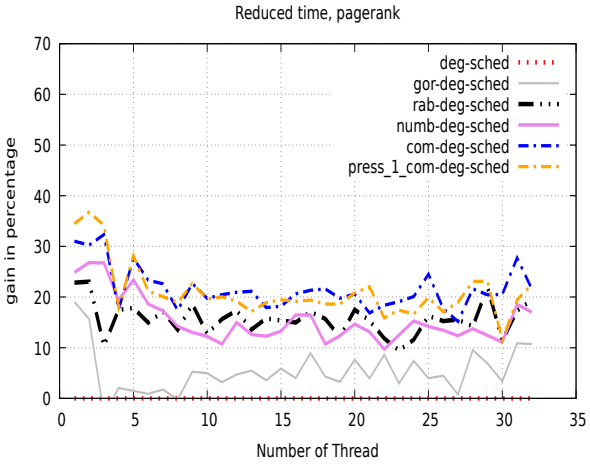

Figure 27: time with scheduling – Okut

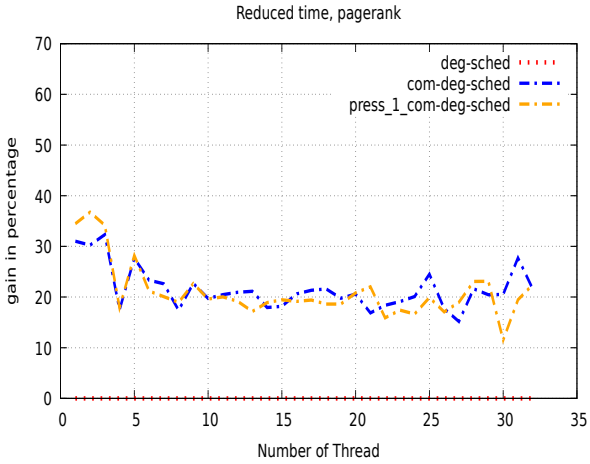

Figure 28: time reduced with scheduling – Orkut

VI CONCLUSION AND FUTURE WORK

Our previous work [17, 18, 20] show how to use some complex-network properties to improve graph applications performance, by a proper memory management (Cn-order) and an appropriate thread scheduling (comm-deg-scheduling). This paper extends this work by combining it with a IDs logarithmization proposed by Besta et al. [19]. Experiments made on one Grid'5000 node showed that this combination gives better results compared to the previous one. For example, we reduce cache-misses from 37.87% to 51.90% (almost ***+14%***) and hence execution time from 18.93% to 28.66% (almost ***+10%***).

We saw in this paper that IDs logarithmization with local approach doesn't give good results experimentally in contrast with its theoretical definition. A direct perspective will be to make experiments fit with theoretic. We are convinced there is still a lot to do in graph ordering for cache misses reduction. For example, on can study the usage of machine learning algorithms in graph ordering.

ACKNOWLEDGEMENTS

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

[1] M. R. Garey, D. S. Johnson, and L. Stockmeyer. "Some simplified NP-complete graph problems". In: *Theoretical computer science* 1.3 (1976), pages 237–267.

[2] S. C. Eisenstat, M. Gursky, M. Schultz, and A. Sherman. *Yale sparse matrix package. I. the symmetric codes*. Technical report. DTIC Document, 1977.

[3] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. "The impact of architectural trends on operating system performance". In: *SIGOPS Operating Systems Review* 29.5 (1995), pages 285–298.

[4] L. Page, S. Brin, R. Motwani, and T. Winograd. "The PageRank citation ranking: bringing order to the web." In: (1999).

[5] M. E. Newman. "The structure and function of complex networks". In: *SIAM* 45.2 (2003), pages 167–256.

[6] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. "Challenges in parallel graph processing". In: *Parallel Processing Letters* 17.01 (2007), pages 5–20.

[7] F. Song, S. Moore, and J. Dongarra. "Feedback-directed thread scheduling with memory considerations". In: *16th international symposium on HPDC*. ACM. 2007, pages 97–106.

[8] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos. "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation". In: *Georgia Institute of Technology, Tech. Rep* (2009).

[9] F. Song, S. Moore, and J. Dongarra. "Analytical Modeling for Affinity-Based Thread Scheduling on Multicore Platforms". In: *Symposim on Principles and PPP* (2009).

[10] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. "STINGER: High performance data structure for streaming graphs". In: *HPEC* (2012), pages 1–5.

[11] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. “Green-Marl: a DSL for easy and efficient graph analysis”. In: *SIGARCH Computer Architecture News*. Volume 40. 1. ACM. 2012, pages 349–362.

[12] J. Riedy, D. A. Bader, and H. Meyerhenke. “Scalable Multi-threaded Community Detection in Social Networks”. In: *IEEE Computer Society Washington, DC, USA 2012* 18.1 (2012). IPDPSW ’12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pages 1619–1628.

[13] D. Nguyen, A. Lenharth, and K. Pingali. “A Lightweight Infrastructure for Graph Analytics”. In: *Proceedings of ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania, 2013, pages 456–471. ISBN: 978-1-4503-2388-8.

[14] Y. Lim, U. Kang, and C. Faloutsos. “SlashBurn: Graph Compression and Mining beyond Caveman Communities”. In: *IEEE Transactions on Knowledge and Data Engineering* 26.12 (2014), pages 3077–3089.

[15] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. “Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis”. In: *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE. 2016, pages 22–31.

[16] H. Wei, J. X. Yu, C. Lu, and X. Lin. “Speedup Graph Processing by Graph Ordering”. In: *Proceedings of the 2016 ICMD*. ACM. 2016, pages 1813–1828.

[17] 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).

[18] T. M. Nguélé, M. Tchuente, and J. 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.

[19] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler. “Log (graph) a near-optimal high-performance graph representation”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pages 1–13.

[20] T. M. Nguélé. *DSL for Social Network Analysis On Multicore Architecture*. Sept. 2018.

[21] R. Barik, M. Minutoli, M. Halappanavar, N. R. Tallent, and A. Kalyanaraman. “Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation”. In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 2020, pages 240–251.

[22] B. Coleman, S. Segarra, A. Shrivastava, and A. Smola. “Graph Reordering for Cache-Efficient Near Neighbor Search”. In: *arXiv preprint arXiv:2104.03221* (2021).

[23] M. K. Esfahani, P. Kilpatrick, and H. Vandierendonck. “How Do Graph Relabeling Algorithms Improve Memory Locality?” In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pages 84–86.