Using Complex-Network properties For Efficient Graph Analysis

Thomas Messi Nguélé, Maurice Tchuente, Jean-François Méhaut

Using Complex-Network properties For Efficient Graph Analysis

Thomas Messi Nguélé, Maurice Tchuente, Jean-François Méhaut

March 30, 2017

Abstract

Complex networks are set of entities in a relationship, modeled by graphs where nodes represent entities and edges between nodes represent relationships. Graph algorithms have inherent characteristics, including data-driven computations and poor locality. These characteristics expose graph algorithms to several challenges; this is because most well studied (parallel) abstractions and implementation are not suitable for them. This work shows how we use some complex-network properties, including community structure and heterogeneity of node degree, to tackle one of the main challenges: improving performance, by improving memory location and by providing proper thread scheduling. In this paper, we firstly formalize complex-network ordering for cache misses reducing as a well known NP-Complete problem, the optimal linear arrangement problem; we then propose cn-order an heuristic that outperforms very recent graph orders. Secondly, we formalize degree-aware scheduling problem as another well known NP-Complete problem, the multiple knapsack problem; then we propose two degree-aware heuristics to solve it. We finally validate our theoretical observations with experiments on a 32 cores NUMA machine with some graph algorithms and some stanford graph datasets. For example, some results with Pagerank algorithm and livejournal dataset show that using cn-order improves performance by reducing cache misses and hence time by 41%; and when cn-order is combined with degree-aware scheduling, time is reduced by 50% due to load balancing among threads.

1 Introduction

A complex network (CN) is a set of entities (individuals, organizations) in a relationship (through friendship or message passing). CNs are modeled by graphs where nodes represent entities and edges between nodes represent relationships. CNs exhibit many properties, among them we have community structure, low density of edges and heterogeneity of node degrees [19]. Due to the huge amount of data, CN algorithms (graph analysis algorithms) are often compute-bound and require high-performance computers to extract knowledge and understand the behavior of entities and their relationships in reasonable time. This paper addresses one of the major challenges of graph analysis [17]: **improving performance**. We are doing it here through memory locality and threads scheduling.

One way to improve graph analysis program performance is through memory locality. In different domains, several studies show that memory operations take most of the time on modern computers: it is shown in [1] that a lot of time is wasted in CPU cache latency (with database query processing); in [4] (with SPEC2000 programs) and in [23] (with on-line transaction processing), the processor is stalled 50% of the time due to cache misses. Thus, writing a program that uses efficiently caches is often a challenge that, if met, increases program performances. This challenge is much visible in graph analysis programs because of poor of locality, which results by the fact that relationships represented by graphs are often irregular and unstructured [17].

Another way to improve graph analysis performance is by ensuring that threads which are working together are well scheduled. Several studies show that improving thread scheduling can reduce time taken by applications: for example, it is shown in [24, 25] that time is reduced more than 25% due to a proper thread scheduling. In graph analysis applications with multiple threads, workload usually consists on given to each thread a same number of nodes (total number of graph nodes divided number of threads). Due to heterogeneity of nodes degree in complex network, this workload can induce to unbalancing load.

1.1 Our contributions

We propose here to improve graph analysis performance by exploiting CN properties: - community structure to tackle memory locality and - heterogeneity of nodes degree to tackle thread scheduling. More specifically, our contributions are as follows:

1. to tackle memory locality,
    - we formalize graph ordering problem for cache misses reducing as optimal linear arrangement problem which is known as NP-complete;
    - we present most recent graph ordering heuristics including gorder, rabbit order and NumBaCo order;
    - we present CN-order, a new graph order which combines advantages of the previous orders;
2. to tackle thread scheduling,
    - we formalize degree-aware scheduling problem as multiple knapsack problem which is known as NP-complete;
    - we present deg-scheduling, a heuristic to solve this problem;
    - we present comm-deg-scheduling, an improved version of deg-scheduling that takes into account graph order in scheduling;
3. to validate theoretical observations,
    - we present experimental results carried out on a 32 cores NUMA (non uniform memory access) machine, with graph analysis algorithm such as Pagerank and Katz score, running on stanford data graphs.

1.2 Outline

The remainder of this paper is structured as follows. Section 2 presents the background that consists of graph analysis challenges, cache management and graph data structures. Section 3 shows theory about graph ordering for efficient graph analysis problem and heuristics to solve this problem. Section 4 presents theory about degree-aware scheduling problem and heuristics designed to solve it. In other to validate theoretical observations, we present experimental results in section 5. Section 6 is devoted to related work and we end with the conclusion and future work in section 7.

2 Background

In this section, we first present graph analysis challenges, then we present how cache memory is managed and we finish with graph data structures.

2.1 Graph Analysis challenges

Graph analysis also called graph algorithms have inherent characteristics [17]:

- computations performed by a graph algorithm are dictated by the vertex and edge structure of the graph on which it is operating rather than being expressed in code;
- data in graph algorithms are unstructured and irregular, that makes the computations and data access tend to have poor locality.

These characteristics make most parallel abstractions and implementation not proper graph algorithms. The main challenges [17, 10, 16] met in graph analysis are:

- capacity: dataset do not fit into a single physical memory;
- performance: due to previous mentioned characteristics, it is not always easy to develop an efficient implementation of a given algorithm. We call by efficient implementation the one that produces the least execution time.

This paper tackle performance challenge and focus in the case when graphs in memory.

2.2 Cache Management

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 which include - 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.

When there is a cache miss, one of the classical algorithms is executed to bring the missed data in the cache: − **optimal algorithm**, the cache line which will not be used for the greatest period of time is replaced, − **random algorithm**, − **LRU** *Least Recently Used*, − **FIFO** *First In First Out*, –**LFU** *Least Frequently Used*, ...

Since general purpose processors (Intel, AMD, ARM) implement one of these algorithms in their hardware, to benefit from all the efficiency of the cache memory, one should make sure that its data structures (graph in our case) are well organized during programs execution.

2.3 Graph representation

Consider the undirected weighted graph with 8 nodes in Fig. 1.

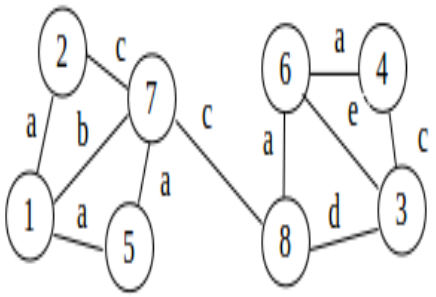

Figure 1: Example of graph (G1)

2.3.1 Matrix Representation

A simple way to represent this graph is to use a matrix representation M where $M(i, j) = m_{ij}$ is the weight of the edge between i and j. This is not very suitable for social graphs because the resulting matrices are sparse. For the graph of Fig. 1, the total used space is 8X8 = 64. But 42 i.e 66% are wasted to save zeros.

2.3.2 Yale representation

The representation often adopted by some graph specific languages such as Galois [20] and Green-Marl [10], is that of Yale [8]. This representation uses three vectors:

- a vector A representing the edges, each edge being represented by one of its ends,
- another vector JA that gives the other extremity of each of the edges of A,
- and a last vector IA which gives the index in vector A of the first nonzero element of each row of the simulated matrix.

The 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)

2.3.3 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 to:

- a block of its neighbors [22] (see Fig. 2a)-);
- a linked list of blocks (with fixed size) of its neighbors, adapted to the dynamic graphs, used by the Stinger platform in [7, 3] (see Fig. 2b)-).

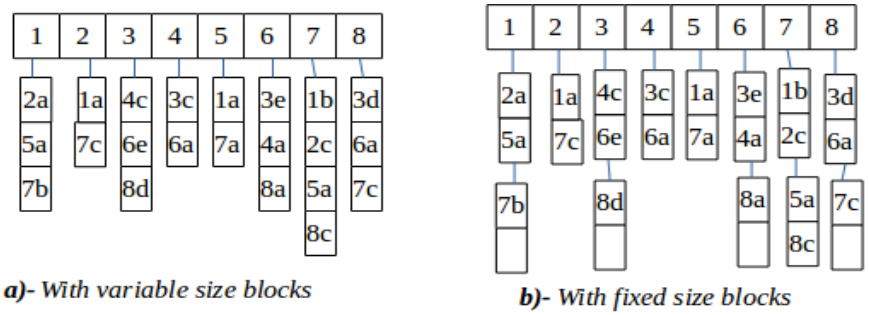

Figure 2: Adjacency list representations of (G1)

3 Graph ordering for efficient graph analysis

3.1 Key idea

Common pattern in graph analysis: One common statement used in graph analysis is as follow:

```
1: for u ∈ V(G) do
2:   for v ∈ Neig(u) do
3:     program section to compute/access v
4:   end for
5: end for
```

With this pattern, we can see that, to reduce cache misses, one should make sure that successive v are closed in memory.

Running 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 $Neig(1) = \{10, 13\}$. The accessing sequence is: $N_6 = (\mathbf{0}, 4, 11, \mathbf{1}, 10, 13)$. With the data cache provide in figure 4,

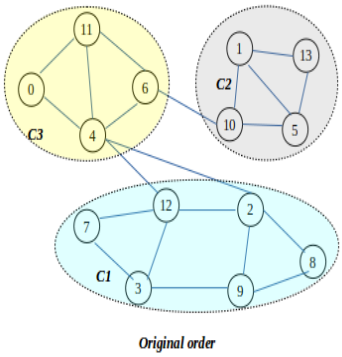

Figure 3: Running example

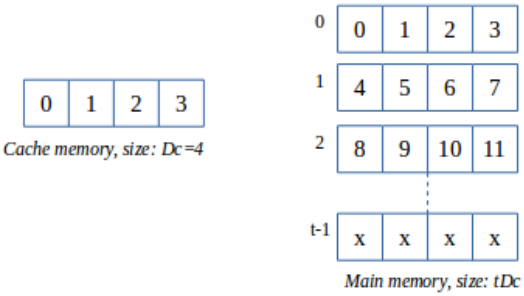

Figure 4: Memory representation

this will cause 6 cache misses (each access of a node will cause a cache miss). But if we consider one of the orders of the same graph got at figure 6, Rabbit order for example, the number of cache misses is reduced. In fact, access of the two first nodes 0 with $Neig(0) = \{1, 3\}$ and 1 with $Neig(1) = \{0, 2, 3, 6, 7\}$ will give the following sequence: $N_9 = (\mathbf{0}, 1, 3, \mathbf{1}, 0, 2, 3, 6, 7)$. This sequence produces only two cache misses; this is because consecutive nodes are closed in memory.

3.2 Problem formalization

3.2.1 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 consider also a memory cache with one line and t blocks in main memory (see figure 4).

Let

- D_c: cache memory size,
- N: set of nodes,
- π: a permutation among N,
- b: $b(x) = x$ **Div** D_c, b is a function that gives the belonging block of a node x in memory (**Div** is the integer division).

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;
- if x and y are not in the same memory block, there is a cache miss.

A cache miss can be modeled by σ as follow:

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

3.2.2 Problem complexity

Let P be a program on a graph $G = (N, E)$. P makes reference to an ordered sequence of nodes $N_k = (n_1, n_2, n_3, ..., n_k)$. The number of cache misses caused by the execution of P is given by:

$$CacheMiss(N_k) = 1 + \sum_{i=2}^{k} \sigma(n_i, n_{i-1})$$

We are looking for a permutation π of G's nodes in such away that the execution of P produces the minimum number of cache misses (min_{dc}). In other words,

$$Let\ G = (N, E), min_{dc} \in \mathbb{N}, \begin{cases} - & Find \quad \pi : \ \mathbb{N} \ \rightarrow \ \mathbb{N} \\ & \qquad\quad\ \ n \ \mapsto \ \pi(n) \\ - & such\ that\ \sum_{i}^{k} \sigma(\pi(n_i), \pi(n_{i-1})) \leq min_{dc}, \\ & with\ \ N_k = (n_1, ..., n_k)\ and\ n_i \in N. \end{cases}$$

Theorem 3.1 (complexity of the problem) *The Numbering Graph Problem (NGP) to minimize the number of cache misses is NP-complete.*

Proof:*One can show it with the Optimal Linear Arrangement Problem (OLAP) known as NP-Complete [9]. As reminder, this problem is defined as follow:*

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

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.

3.3 Graph ordering heuristics

In this section, we first present three recent algorithms: Gorder [26], Rabbit Order [2] and NumBaCo [18]; These three algorithms were published quite at the same period and each of them claims to outperform state of the art algorithms. Then, we present a new algorithm which combine the strengths of all the previous ones.

3.3.1 Gorder

Gorder idea: The goal is to reduce cache misses by making sibling nodes be close in memory (allow them to fit in cache memory). Consider the graph in figure 3. $Neig(1) = \{5, 10, 13\}, Neig(5) = \{1, 10, 13\}$. Let consider the data cache provided in figure 4.

From figure 3, we can see that if the graph algorithm accesses $Neig(1)$ with subsequence $N_4 = (\mathbf{1}, 5, 10, 13)$ it will cause 4 CPU cache misses; with $Neig(5)$ and $N_4 = (\mathbf{5}, 1, 10, 13)$, it will cause 4 CPU cache misses.

From figure 6, with gorder, $\pi(1) = 5$ and $\pi(5) = 6$; access to $Neig(5) = \{4, 6, 7\}$ with subsequence $N_4 = (\mathbf{5}, 4, 6, 7)$ will cause 1 CPU cache miss and access to $Neig(6) = \{4, 5, 7\}$ with $N_4 = (\mathbf{6}, 4, 5, 7)$ will cause 1 CPU cache miss. This example shows that Gorder allows to reduce cache misses.

Goder key function: Gorder tries to find a permutation π among all nodes in a given graph G by keeping nodes that will be frequently accessed together in a window w, in other to minimize the cpu cache miss ratio.

Hao Wei and co-authors [26] defined a score function $S(u,v) = S_s(u,v) + S_n(u,v)$, where $S_s(u,v) = |N_{in}(u) \cap N_{in}(v)|$ is the number of times u and v co-exist in sibling relationship, the number of their common in-neighbors; $S_n(u,v)$ is the number of times that u an v are in neighbor relationship, which is 0, 1 or 2 since (u,v) and (v,u) may co-exist in a direct graph.

Based on this score function , the problem is to maximize the locality of two nodes to be placed closely. And this is to find a permutation π that maximize the sum of score $S(.,.)$, where $\pi(u)$ assigns every u with an unique number $[1,n]$, $n = |N|$ is the number of nodes.

$$Let\ G = (N,E), \begin{cases} - & Find \quad \pi : \ \mathbb{N} \ \rightarrow \ \mathbb{N} \\ & \qquad\quad\ \ v \ \mapsto \ \pi(v) \\ - & such\ that \ \ F(\pi) = \sum_{i=1}^{n} \sum_{j=max\{1,i-w\}}^{i-1} S(v_i, v_j), \\ & is\ maximal. \end{cases}$$

Gorder algorithm: Algorithm 1 presents Gorder. Hao Wei and co-authors [26] showed that this algorithm is $\frac{1}{2w}$-approximation and with an appropriated priority queue, it is $O(\sum_{u \in V}(d_o(u))^2)$ time complexity.

Consider the last nodes inserted in P: $v_{i-w}, ..., v_{i-2}, v_{i-1}$. The next v_i to be inserted is chosen in such away that $\sum_{j=max\{1,i-w\}}^{i-1} S(v_i, v_j)$ is maximal.

Algorithm 1 : **Gorder**

Input: $G = (V,E), w, S(.,.)$
Output: π a permutation, $P[i] = x$ means $\pi(x) = i$

1: Select a node v as the start node $p[1] \leftarrow v$
2: $V_r \leftarrow V(G) - \{v\}, i \leftarrow 2$
3: **while** $i \leq n$ **do**
4: $v_{max} \leftarrow \emptyset$, $k_{max} \leftarrow \infty$
5: **for** $v \in V_r$ **do**
6: $k_v \leftarrow \sum_{j=max\{1,i-w\}}^{i-1} S(P[j], v)$
7: **if** $k_v > k_{max}$ **then**
8: $v_{max} \leftarrow v$, $k_{max} \leftarrow k_v$
9: **end if**
10: **end for**
11: $P[i] \leftarrow v_{max}$, $i \leftarrow i + 1$
12: $V_r \leftarrow V(G) - \{v_{max}\}$
13: **end while**

Let consider the problem of finding frequent itemsets, itemsets are set of neighbors; choose the best v_i can be seen as choose the frequent itemset of size 2 between $\{v_{i-w}, v_i\}, ...\{v_{i-2}, v_i\}, \{v_{i-1}, v_i\}$. It is interesting to know how will be performances if we consider frequent itemsets with size more than 2.

Advantage and disadvantage: The main advantage of gorder is that it brings closer pairs of nodes appearing frequently in direct neighborhood. But the main disadvantage is that it doesn't care about the community structure which is usually present in real graphs. During the numbering, nodes that

belong to same community may be scattered. This is for example the case in Appendix B with karate graph.

3.3.2 NumBaCo and Rabbit Orders

The main idea of these orders is to make nodes that belong to the same community (for NumBaCo) or to the same sub-community (for Rabbit) to be consecutive. In that way, nodes belonging to the same community or sub-community will be close in memory.

Both NumBaCo and Rabbit orders start by detecting communities in a graph before numbering. If NumBaCo uses Louvain algorithm [5], Rabbit order uses a light version of Louvain (in other to insure just in time ordering).

Community detection with Louvain: Louvain algorithm [5] is one of the most popular community detection heuristic. It is based on a quality function called modularity, that assigns to a partition a scalar value between -1 and 1, representing the density of links inside communities as compared to links between communities. It starts with a partition where each node is alone in its community. Then, it computes communities by repeating iteratively the two following phases:

> ***Step 1)*** for each node i, evaluate the gain in modularity that may be obtained by removing i from its current community and placing it in the community of a neighbor j. Place i in the community for which this gain (if positive) is maximum.
>
> ***Step 2)*** Generate a new graph where nodes are communities detected in the first phase. Reapply the first phase of the algorithm to the resulting weighted network.

These two phases are iterated until the maximum modularity is reached.

Louvain algorithm generates a merged structure corresponding to lower level and higher level group structures: the lowest level, level one contains structures got at the first iteration of the two phases and the highest level, last level contains structures got at the last iteration.

Communities generated by graph in figure 6, is given at figure 5. Level one has 4 communities {0,1,2,3}, level two (the last) has 3 communities {0,1,2}. Another example is shown in Appendix A with karate graph.

Rabbit order: Algorithm 2 present Rabbit order. Line 1 generates an hierarchical set of communities: that is, each community contains sub-communities which may be divided in sub-sub-communities and so on. Junya Arai and co-authors [2] propose a parallel community detection, that allows them to ensure just in time numbering.

Algorithm 2 : **Rabbit Order**

Input: $G = (V, E)$
Output: π a permutation on V nodes
1: $Com \leftarrow detect\_comm(G)$
2: $\pi \leftarrow graph\_numbering(Com, G)$
3: **return** π

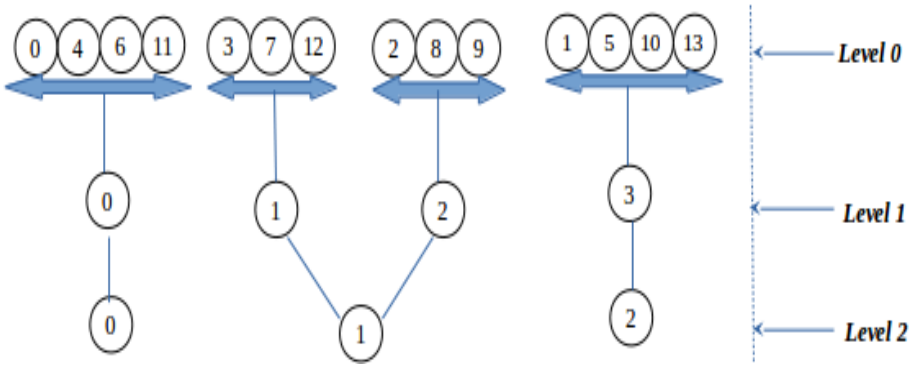

Figure 5: Running graph with Louvain

Line 2 of Rabbit order visits each community in DFS fashion and assigns a new number to each node. This action makes nodes belong to the same sub-community be close in the memory. Figure 6 gives rabbit order of graph in figure 3. Nodes that belong to the same sub-community are consecutive, for example $\pi\{0, 4, 6, 11\} = \{0, 1, 2, 3\}$.

NumBaCo: The first line of NumBaCo generates communities with Louvain algorithm. The second line classifies communities to make ones with higher affinity (number of edge shared) be together. Line three assigns numbers in such away that nodes belong to the same communities are consecutive. Community j follows community i if community j has the highest affinity for all j $>$ i. Before generate a new graph at line 5, line 4 first change neighbors position according to their appearance in the hierarchy: first store neighbors that are in level 1, then level two and so on. The total complexity of the algorithm

Algorithm 3 : **NumBaCo** (**Num**bering **Ba**sed on **Co**mmunities)

Input: $G = (V, E)$
Output: $G' = \pi(G)$, π is a permutation, neighborhood in G' is stored hierarchically

1: $Com \leftarrow detect\_comm\_Louvain(G)$
2: $Com_{cl} \leftarrow classify\_comm(Com)$
3: $\pi \leftarrow graph\_numbering(Com_{cl}, G)$
4: $G_1 \leftarrow store\_neighboors(Com, G)$
5: $G' \leftarrow new\_graph(G_1, \pi)$
6: **return** G'

is $O(nlogn + nkc + n(k + 1) + C^2)$. Where n is number of nodes, k is the mean degree, C the number of communities at last level and c is the number of communities at level one.

Differences between NumBaCo and Rabbit: In addition to the two steps, including communities classification and neighborhood storage, NumBaCo and Rabbit are different in the numbering. While NumBaCo emphasis in the last level community, Rabbit ensure that nodes of the first level community have consecutive numbers. For example, in figure 3, $\pi\{2, 3, 7, 8, 9, 12\}$ gives

$\{4, 5, 6, 7, 8, 9\}$ in figure 6 with NumBaCo order and $\{7, 4, 5, 8, 9, 6\}$ with Rabbit order.

To get the benefit of both NumBaCo and Rabbit order, one can change the emphazing level. That consists on modifying line 2 of NumBaCo algorithm (refers as NumBaCo-Rabbit in table 3). Instead of classifying the last level community (as in NumBaCo), we classify level 2. In fact, $classify\_comm(com)$ takes C^2 where C is $|com|$. Level 1 of Louvain algorithm can generate many sub-communities (more than $\sqrt{n}$). To avoid spending time at this step, we classify sub-communities level 2 (which has less elements than level 1 but much elements than last level).

Advantage and disadvantage: The main advantage of NumBaCo or Rabbit is that they allow nodes belonging to the same community or sub-community to be closer in memory. The disadvantage is that the numbering within a community or sub-community is random. May be, a good numbering within communities can allow better performances.

3.3.3 Strengths and weaknesses of each heuristic

Table 2: Strengths and weaknesses

heuristics	Gorder	Rabbit	NumBaCo
strengths	Bring closer sibling nodes	Bring closer sub-communities nodes	Bring closer comm nodes and communities
weaknesses	scatter communities or sub-communities	- sub-communities can be big - order in sub-communities is improvable	- communities. can be big - order in communities is improvable

Table 2 groups strengths and weaknesses of the three studied order heuristics. The next paragraph shows how we use these strengths and weaknesses to build a more powerful order.

3.3.4 A new complex-network ordering (CN-order)

CN-order combines the advantages of all previous studied algorithm (see algorithm 4 below).

Algorithm 4 : **C**omplex **N**etwork order (**CN-order**)

Input: $G = (V, E)$
Output: $G' = \pi(G)$, π is a permutation, neighborhood in G' is stored hierarchically

1: $\pi_1 \leftarrow Gorder(G)$
2: $Com \leftarrow detect\_comm\_Louvain(\pi_1(G))$
3: $Com_{cl} \leftarrow classify\_comm(Com)$
4: $\pi \leftarrow graph\_numbering(Com_{cl}, \pi_1(G))$
5: $G_1 \leftarrow store\_neighboors(Com, \pi_1(G))$
6: $G' \leftarrow new\_graph(G_1, \pi)$
7: **return** G'

With CN-order,

- Sibling nodes are kept close,

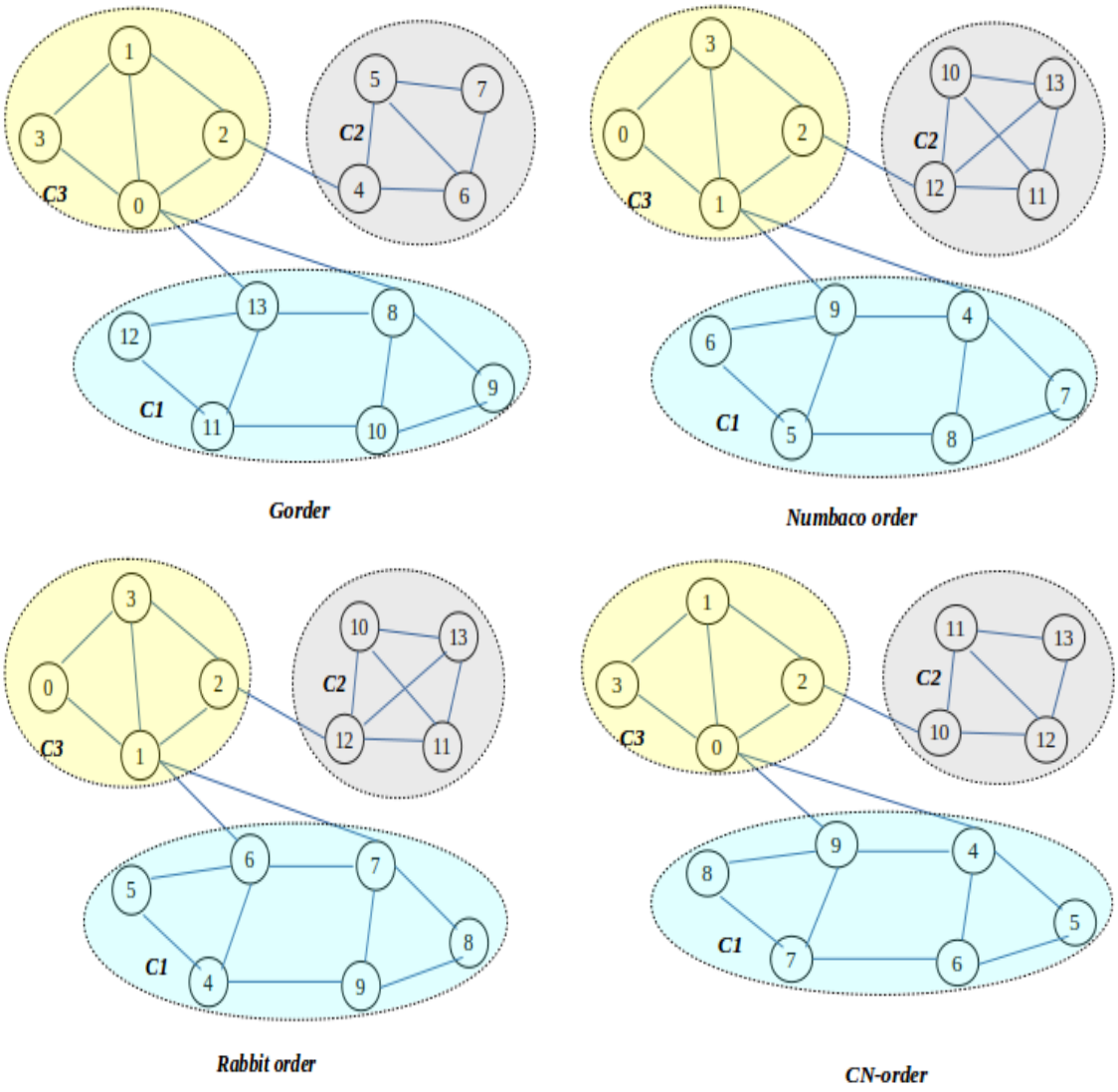

Figure 6: Example of graph with different orders

- Nodes that belong to same community are close,
- And communities with higher affinity are close.

The main disadvantage of CN-order is in its time complexity: gorder time complexity + numbaco time complexity. This issue can be tackled by parallelizing cn-order. We leaf this problem as future work.

4 Degree-aware scheduling for load balancing

4.1 Scheduling problem

Description and idea: In most graph analysis parallel programs with multiple threads (resulting from graph systems such as Galois [20] or Green-Marl [10] for example), it was observed that, each thread is in charge of groups of nodes. In that way the workload of a thread depends of the number of nodes explored by this thread; this number is usually the same for all the threads, $\frac{n}{nb\_thread}$ (where n is the number of nodes and nb_thread is the number of threads).

Consider the common pattern presented at section 3.1, it is easy to note that the time spent on a node u depends on $|Neig(u)|$. This can lead to an unbalanced load in applications processing on real graphs which usually have nodes with heterogeneous distribution of degrees. This is for example the case with social graphs which have "famous" nodes i.e. nodes with a very high degree compared to the others.

The idea is to take into account the heterogeneity of node degrees of real graphs to ensure load balancing in loop scheduling of graph analysis.

Formal definition and complexity: Given a graph G = (N, E) and a parallel graph analysis program P with t threads running on G, the problem is to know how to assign nodes to threads in other to ensure balancing load among threads. Let us denote:

- $d_i = |Neig(i)|$ degree of node i,
- $T = \{th_1, th_2, ..., th_t\}$ set of threads,
- x_{ij} a decision variable,

$$x_{ij} = \begin{cases} 1 & if\ node\ i\ will\ be\ processed\ by\ th_j \\ 0 & else \end{cases}$$

Let P with t threads on $G = (N, E)$,

$$Problem:\ d_{max} = \frac{\sum_{i=1}^{n} d_i}{t} \begin{cases} maximize\ s_j = \sum_{i=1}^{n} x_{ij}, \sum_{j=1}^{t} s_j = n \\ subject\ to\ \sum_{i=1}^{n} x_{ij} d_i \leq d_{max}, \\ with : x_{ij} = \{0, 1\}, \sum_{j=1}^{t} x_{ij} \leq 1 \\ and\ i = \{1, ..., n\}, j = \{1, ..., t\} \end{cases}$$

This definition makes it easy to see this problem as **multiple knapsack problem** [15] which is NP-hard. Thus trying to schedule threads by ensuring load balancing with nodes degree is NP-hard. We propose in the following section heuristics for this problem.

4.2 Scheduling heuristics

Before presenting scheduling heuristics, we are going to show the parallel version of common pattern presented at section 3.1.

4.2.1 Parallel version of common pattern

Here is a task pattern that will be run by a thread:

```
1: task_func(s,e)
2: for u ∈ {s, ..., e} do
3:    for v ∈ Neig(u) do
4:       program section to compute/access v
5:    end for
6: end for
```

There are two cases depending of the number of tasks compare to the number of threads. If there are equal, then each thread is attached to one task: the scheduling is static. But if there are not equal ($nb_{thread} < nb_{task}$), then one thread can be attached to more than one thread: the scheduling is dynamic.

Common pattern for static scheduling: Each thread th_t (spawned at line 3) will get exactly one task which corresponds to a slice of nodes from $task\_load[t].start$ to $task\_load[t].end$ on which the thread is processing.

```
1: for t ← 1 to nb_thread do
2:    s ← task_load[t].start; e ← task_load[t].end
3:    spawn_thread(th_t, task_func, param(s, e))
4: end for
```

Common pattern for dynamic scheduling: In this case, each thread th_t may have more than one task $t \in setOfTask$.

```
1: global setOfTask = {1, ..., t}
2:
3: do_work()
4: while setOfTask ≠ ∅ do
5:    t ← atomic_take_task(setOfTask)
6:    s ← task_load[t].start; e ← task_load[t].end
7:    task_func(s, e)
8: end while
9:
10: for t = 1 to nb_threads do
11:    spawn_thread(th_t, do_work())
12: end for
```

4.2.2 Scheduling

The goal of the following heuristics is to find the workload of each thread. This workload is based on nodes degree.

Degree-aware scheduling: This algorithm builds a set of tasks that will be assigned to threads. Each task is a set of consecutive nodes. The number of nodes present in this set depends on d_{max}, the maximum degree that should have each task. The algorithm greedily adds node i to task t nodes collection until $\sum d_i$ reaches d_{max}.

Algorithm 5 : **deg-scheduling** (**Deg**ree-aware **scheduling**)

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

1: $d_{max} \leftarrow \frac{d_{sum}}{nb_{task}}$, $t \leftarrow 0$, $i \leftarrow 0$
2: **while** $i < n$ **and** $t < nb_{task}$ **do**
3: $task[t].start \leftarrow i$, $d_{tmp} \leftarrow 0$
4: **while** $d_{tmp} < d_{max}$ **and** $i < n$ **do**
5: $d_{tmp} \leftarrow d_{tmp} + d_i$
6: $i \leftarrow i + 1$
7: **end while**
8: $task[t].deg \leftarrow d_{tmp}$, $task[t].end \leftarrow i + 1$
9: $i \leftarrow i + 1$
10: **end while**
11: **return** $task[\,]$

The complexity of this algorithm is O(n).

Community and degree aware scheduling: We saw at section 3 that keeping nodes in good ordering allows to reduce cache misses and hence execution time. Noting the fact that **deg-scheduling** algorithm keeps nodes consecutive for each thread, performance will probably be increased if the graph used for processing has the best order. This observation leads to design the following algorithm 6.

Algorithm 6 :comm-deg-scheduling

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

1: $\pi \leftarrow$ CN-order(G)
2: $task[\,] \leftarrow$deg-scheduling($\pi(G), d_{sum}, nb_{thread}$)
3: **return** $task[\,]$

In this algorithm, we first process CN-order before scheduling. By that, we ensure that nodes processed by each thread will be closed in memory.

5 Experimental evaluation

For this evaluation, we used two graph analysis applications including Katz score and Pagerank.The experiments were made on a machine (idrouille) which has 4 NUMA nodes with 8 cores per node, making a total of 32 cores for 64 GB of memory. Each node is of type Intel Xeon characterized by 2.27 GHz, L1 of 32 KB, L2 of 256 KB, L3 of 24 MB, no Hyper-Threading. Fig. 7 presents a NUMA node.

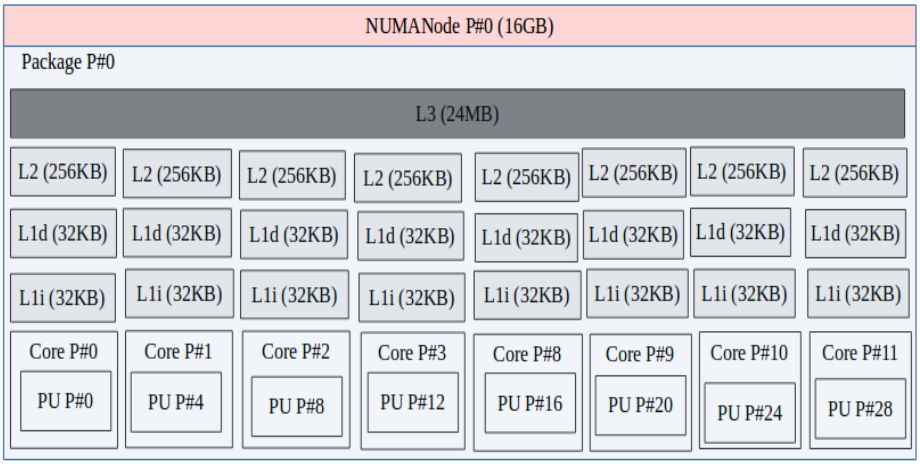

Figure 7: idrouille node: 8 cores, private L1 and L2, shared L3

Pagerank: Pagerank [21] is an algorithm used by Google to classify pages on the web. A Pagerank of a page x is given by the following formula:

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

Where:

- d is the probability to follow this page, $(1-d)$ is the probability to follow another.
- $N_{in}(x)$ is the set of incoming neighbors of x.
- $N_{out}(y)$ is the set of outgoing neighbors of y.

We used a posix thread implementation proposed by *Nikos Katirtzis*[1]. This implementation uses *Bloc* data structure.

Katz score: Katz Score [14] can be used as a similarity measure based on the distances between the nodes. Katz score between two nodes x and y is given by following formula:

$$katz\_score(x,y) = \sum_{l=1}^{L} (\beta^l . |paths_{x,y}^{<l>}|) \tag{2}$$

Where:

- L represents the maximum path size.
- $paths_{x,y}^{<l>}$ is the set of paths of length l between x and y, and $|paths_{x,y}^{<l>}|$ represents its cardinality.

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

- $0 < \beta < 1$. β is chosen such that the paths with a big l contribute lesser to the sum than the paths with a small l.

For any node $x \in G$, it can be shown that the cardinalities of the sets of paths from this node to the other nodes are computed as follow (at each step i, N_i is the set of neighbors and L_i is the set of cardinalities of the paths set):

$$\begin{cases} i = 1 & N_i = Neig(x) \\ & L_i[y] = 1, \forall y \in N_i \\ 2 \leq i \leq L & N_i = \{z / z \in Neig(y) \wedge y \in N_{i-1}\} \\ & L_i[z] = \sum_y L_{i-1}[y]_{/\{y \in N_{i-1} \wedge z \in Neig(y)\}} \end{cases} \tag{3}$$

5.1 Graph ordering comparison

Comparison on a single core: We did this experimentation on idrouille described above (with L1 of 32 KB, L2 of 256 KB, L3 of 24 MB); the dataset is livejournal [27] unoriented graph with 3997962 nodes and 34681189 edges. Graph was represented with an adjacency list which is O(2m + n) space. So the space taken by graph in memory is (2*34681189 + 3997962)*4bytes = 279.84 MB (do not fit in L3 cache).

Table 3: Performances comparison (Pagerank, 1 thread)

Numbering algorithm	Original	Gorder	Rabbit	NumBaCo	Rabbit-NumBaCo	CN-Order
L3 cache-references	8,023M	5,688M	6,224M	7,322M	6,715M	5,561M
Earned %		**29.10%**	22.42%	8.74%	16.30%	**30.68%**
L3 cache-misses	3,569M	2,985M	2,234M	2,014M	2,050M	1,897M
Earned %		16.36%	37.41%	**43.57%**	42.56%	**46.84%**
Execution time	348.516s	305.626s	265.691s	261.099s	259.849s	243.442s
Earned %		12.31%	23.77%	**25.08%**	25.44%	**30.14%**

Results in table 3 confirm the strengths and weaknesses of each algorithm presented in table 2.

- All the three orders outperform the original order.
- Gorder allows to have least references to L3 than Rabbit and NumBaCo. That means it allows to have a higher reference to L1 and L2 (with cache hit). It confirms the fact that Gorder brings sibling nodes close compared to the others.
- NumBaCo has the least cache misses, close to Rabbit order.
- When we combine Rabbit and NumBaCo, we have performances between the both.
- CN-order outperforms all the orders, since it combines all the advantages: 30.68% of cache references, 46.84% of cache misses and 30.14% of execution time.

Comparison on multiple cores: Observations done with one thread (on one core) are quite the same with multiple threads (from 2 to 32). Figure 8 shows the execution time with different orders. In this figure, cn-order curve remains almost below all the other curves, which confirms it is the best order compared to the others. Rabbit, numbaco and rabbit-numbaco curves are switching positions; this can be explained by the architecture used (see Fig. 7), more precisely by the L3 cache which is shared among the cores: cores benefit more and more to data loaded by others. Gorder cure, which is far compared to the last three is also switching its position with original order; this switching is also explained by the architecture.

In figure 9, we see the variations of reduced time with cn-order and rabbit-numbaco order. For example with 32 threads, cn-order reduced time by 41% and rabbit-numbaco order reduced time by 30%.

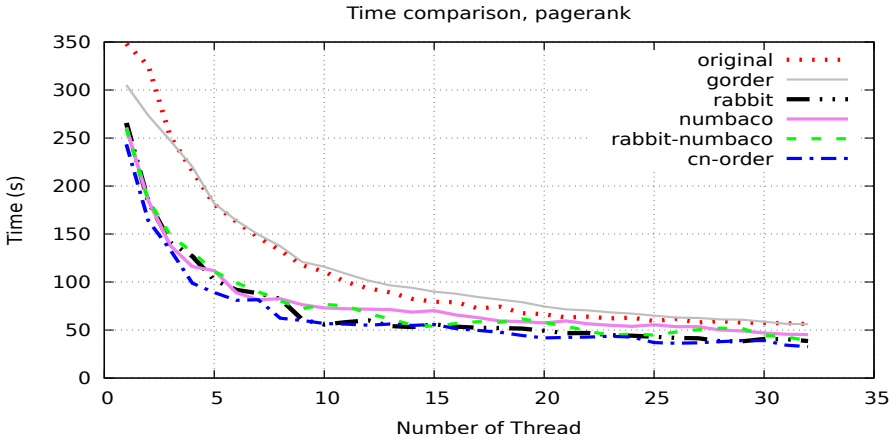

Figure 8: Time comparison with graph orders

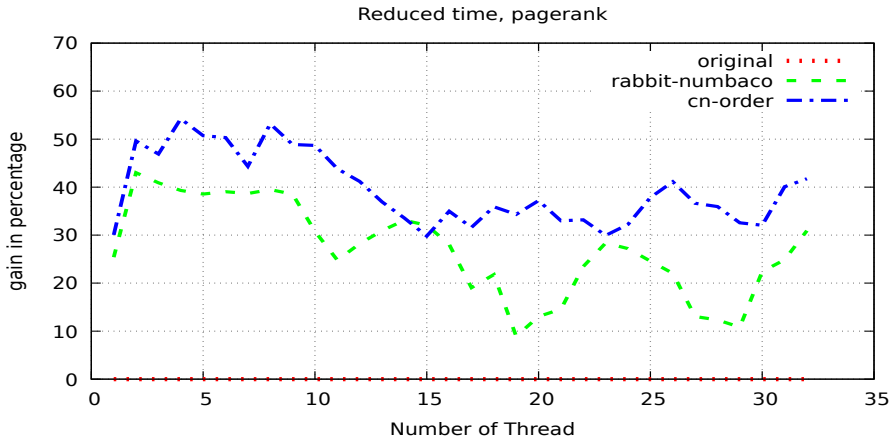

Figure 9: Reduced time with cn-order vs rabbit-numbaco

As already observed with one thread, time is reduced because cache misses

(and cache references) are reduced. Figure 10 shows cache misses with different orders. Note that, curves display the mean number of cache-misses (at each y, we have $\frac{Total\ number\ cache-misses}{number\ of\ threads}$).

Figure 11 shows that with cn-order (the best order), cache-misses are reduced from 25% to 47%. Curves comparing cache-references are shown at Appendix C.

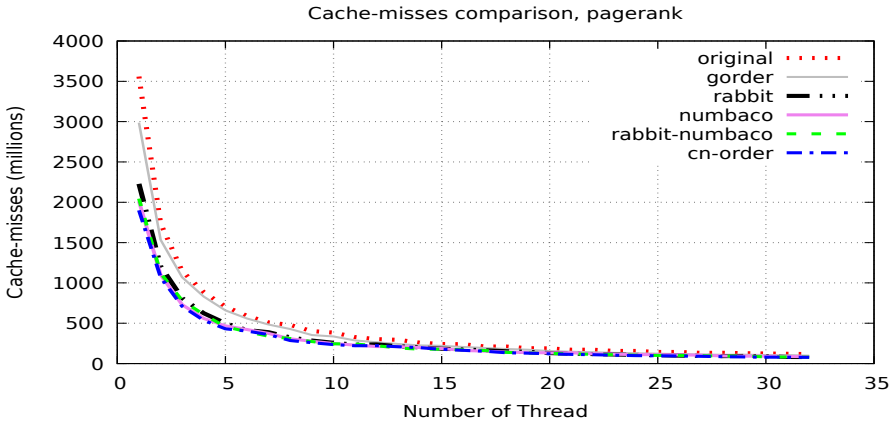

Figure 10: Cache misses comparison with graph orders

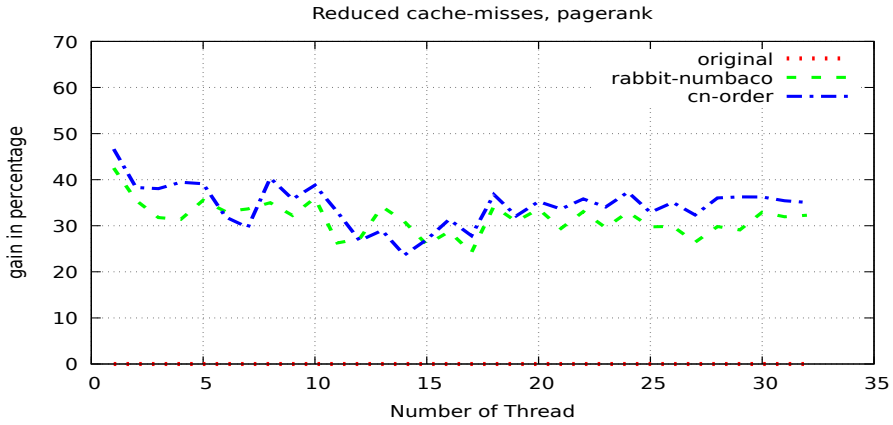

Figure 11: Reduced cache misses with cn-order vs rabbit-numbaco

5.2 Graph ordering and degree-aware scheduling

We show in this section how degree-aware scheduling improves performances by reducing execution time.

5.2.1 Impact of degree-aware scheduling in load balancing

In figure 12, each thread receives $\frac{number\ of\ nodes}{number\ of\ thread}$ nodes as its workload. Due to heterogeneity of nodes degree in complex network, this workload leads to unbalancing load. Some threads take higher time and others smaller time. For example, in figure 12, threads 0, 1, 2, 3 and 4 take more than 5% of total time; and threads 31, 30, 32 take less than 1.5 % of total time. There is a high variation around the mean 3.12% that should take each of 32 threads in case of load balancing. The standard deviation of different times is 1.66. It is interesting to note that time taken by a thread is almost proportional to the sum of degree of nodes it processed. This observation is used in figure 13 to reduce unbalancing load.

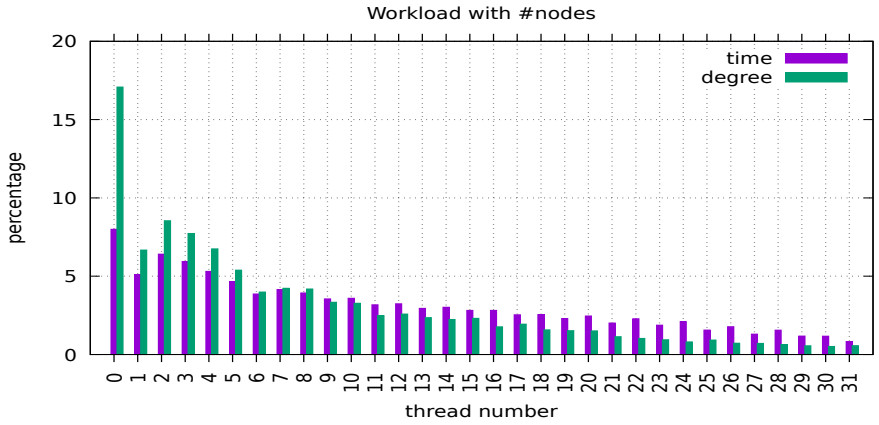

Figure 12: Scheduling nodes

In figure 13, each thread receives $\frac{Total\ of\ nodes\ degree}{number\ of\ thread}$ as it workload. This workload contributes to reduce unbalancing load. Each thread now takes between 1.63% and 4.3 % of time. In this case, the standard deviation is 0.64.

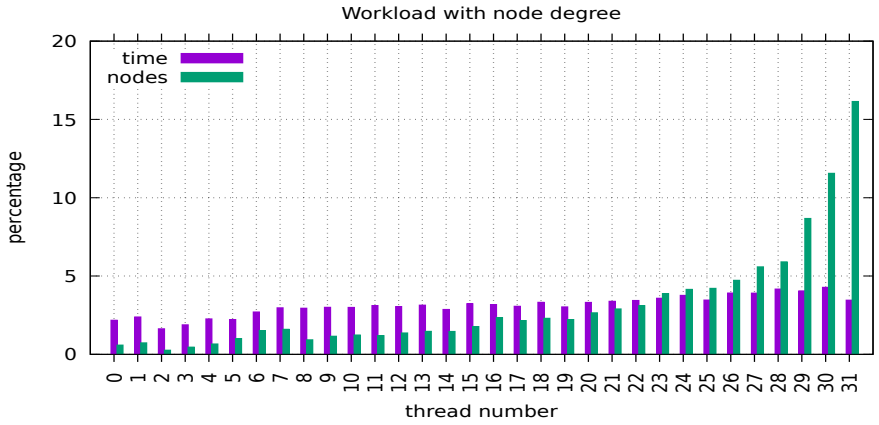

Figure 13: Scheduling degree

5.2.2 Impact of degree-aware scheduling in time reducing

Figures 14 and 15 compare the execution time between scheduling with nodes and scheduling with degree. Consider time obtained with 32 threads:

- compare to node-aware scheduling, with original order, when using degree-aware scheduling, time is reduced by 20%;
- compare to node-aware scheduling, with cn-order, time is reduced from 41% to 50%.

The first observation shows that degree-aware scheduling 'alone' improves performances. And the second observation shows that performances due degree-aware scheduling and performances due to cn-order are also combined.

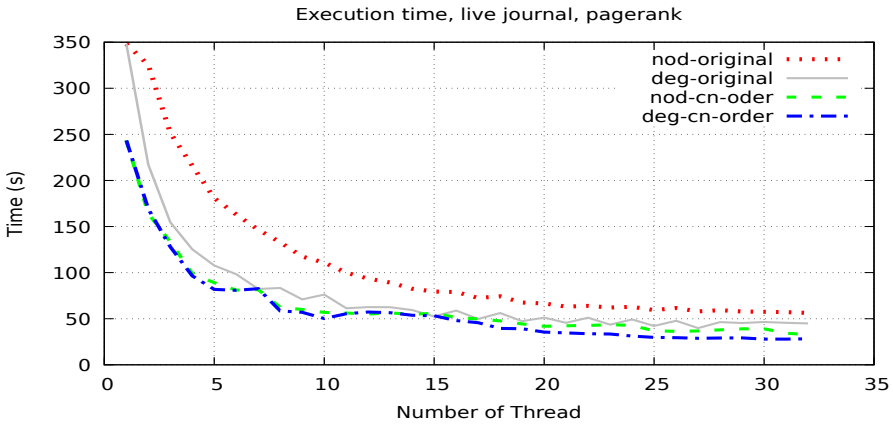

Figure 14: Scheduling with degree (deg) vs scheduling with nodes (nod)

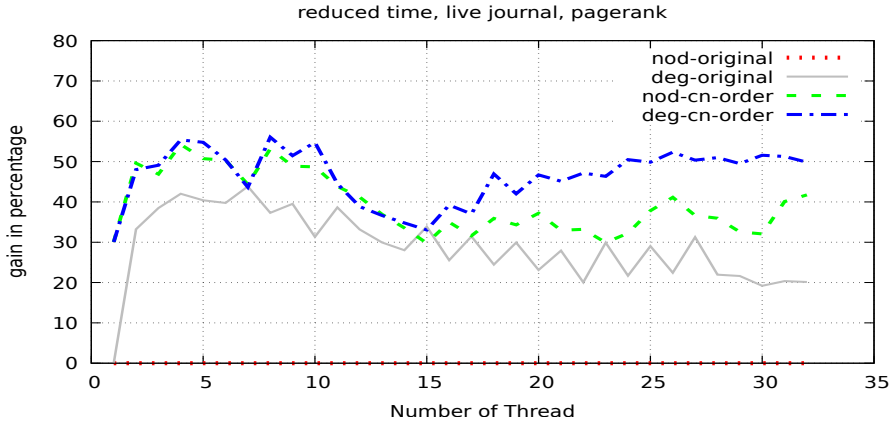

Figure 15: Reduced time due to scheduling and ordering

6 Related work

Graph orders: We have already mentioned most recent graph orders: Gorder [26], Rabbit Order [2] and NumBaCo [18]; each of them claims to outperform state of the art algorithms such as BFS order [12], graph partition order with METIS [13]. We presented in this paper a new graph order that outperform the most recents, and by that way graph orders of the state of the art.

Perfomances and scheduling: In this paper, to ensure load balancing in graph analysis applications, we define degree-aware scheduling as multiple knapsack problem and then propose heuristics to solve it. There are other works using scheduling to improve performances:

- Song and co-authors [24, 25] propose an approach based on memory trace analysis and graph partitioning in other to generate and optimized schedule to run threads on NUMA systems.
- Yenke and co-authors [28] define scheduling of computing services as knapsack problem and propose an algorithm to solve it. This algorithm produces a very small overhead when using on checkpointing process.

Perfomances and community structure: In other to reduce cache misses, we make sure (through CN-order in algorithm 4) that nodes belonging to the same community are close in memory. In other works:

- Duong and co-authors [6] propose an algorithm that takes advantage of the community structure to shard social graphs across a distributed system.
- Hoque and co-authors [11] design an organizational technique of hard drive based on community structure of social graphs data. This technique allowed them to reduce the number of movement of the read head and thus improve disk access (48% faster).

7 Conclusion

In this paper, we showed how we used complex-network properties – community structure and heterogeneity of node degree – to improve graph analysis performance.

We firstly defined complex-network ordering for cache misses reducing as the optimal linear arrangement problem, a well known NP-Complete problem. We then studied the most recent graph ordering heuristics including gorder, rabbit order and NumBaCo order, each of them claiming to outperform state of the art graph orders such as BFS order, graph partitioning order with METIS. This study leads us to design CN-order, a new graph order which combines advantages of the most recent orders. Experimental results on a 32 cores NUMA machine (with Pagerank and livejournal for example) showed that cn-order outperforms the recent orders and compared to original order, cn-order allows to reduce cache-misses and hence execution time by 41%.

We secondly defined degree-aware scheduling problem as multiple knapsack problem which is also known as NP-complete. We first proposed deg-scheduling,

a heuristic to solve this problem; then we proposed comm-deg-scheduling, an improved version of deg-scheduling that takes into account graph order in scheduling. Then experimental results showed that the last heuristic improves time reducing by 9%, making the total time reducing by 50%.

Algorithms presented above are intended to be executed before a graph algorithm, for preprocessing. This is to have an organization of nodes in memory that minimizes cache misses; and also a proper scheduling to ensure load balancing among threads. In this paper, we didn't care about the time taken by proposed heuristics, cn-order and comm-deg-scheduling. In terms of complexity compared to cn-order and comm-deg-scheduling, we distinguish two types of graph algorithms:

- **category 1**: graph algorithms which have higher complexity,
- **category 2**: graph algorithms which have lesser complexity.

Graph algorithms in **category 1** will obviously benefit to our performance improvement. But for graph algorithms in **category 2**, our proposed heuristics should be tuned. This can be done for example by parallelizing heuristics. It is left as future work.

References

[1] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB'' 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number DIAS-CONF-1999-001, pages 266–277, 1999.

[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 22–31. IEEE, 2016.

[3] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[4] Kristof Beyls and Erik H D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *International Conference On Computational Science*, pages 448–455. Springer, 2004.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[6] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. Sharding social networks. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 223–232. ACM, 2013.

[7] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. Stinger: High performance data structure for streaming graphs. *HPEC*, pages 1–5, 2012.

[8] Stanley C Eisenstat, MC Gursky, MH Schultz, and AH Sherman. Yale sparse matrix package. i. the symmetric codes. Technical report, DTIC Document, 1977.

[9] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[10] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[11] Imranul Hoque and Indranil Gupta. Social network-aware disk management. 2010.

[12] Konstantinos I Karantasis, Andrew Lenharth, Donald Nguyen, María J Garzarán, and Keshav Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE Press, 2014.

[13] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[14] Leo Katz. A new status index derived from sociometric analysis. *Psychometrica-vol.18, No.1*, 18(1), March 1953.

[15] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *knapsack problems*. Springer, 2004.

[16] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[17] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[18] Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. Social network ordering to reduce cache misses. *under reviewing in arima, Accept after minor changes (in first round review, in oct 2016)*, 2016. https://hal.archives-ouvertes.fr/hal-01304968.

[19] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[20] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[21] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[22] Jason Riedy, David A. Bader, and Henning Meyerhenke. Scalable multithreaded community detection in social networks. *IEEE Computer Society Washington, DC, USA 2012*, 18(1):1619–1628, 2012. IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

[23] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. *ACM SIGOPS Operating Systems Review*, 29(5):285–298, 1995.

[24] Fengguang Song, Shirley Moore, and Jack Dongarra. Feedback-directed thread scheduling with memory considerations. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 97–106. ACM, 2007.

[25] Fengguang Song, Shirley Moore, and Jack Dongarra. Analytical modeling for affinity-based thread scheduling on multicore platforms. *Symposim onPrinciples and Parctice of Parallel Programming*, 2009.

[26] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[27] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[28] Blaise Omer Yenke, Jean-François Mehaut, and Maurice Tchuente. Scheduling of computing services on intranet networks. *IEEE Transactions on Services Computing*, 4(3):207–215, 2011.

A Karate hierarchy with Louvain

Figure 17 show karate hierarchy.

B Karate with different orders

B.1 Karate with goder

With gorder in figure 18, nodes belonging to the same community in figure 16 do not necessarily have consecutive numbers. For example, node 1 in community **C4** is followed by node 2 which is community **C2**.

B.2 Karate with rabbit order

Figure 18 gives karate graph with rabbit order. Nodes that belong to the same sub-community are consecutive, for example $\pi\{0, 1, 11, 17, 19, 21\} = \{0, 1, 2, 3, 4, 5\}$.

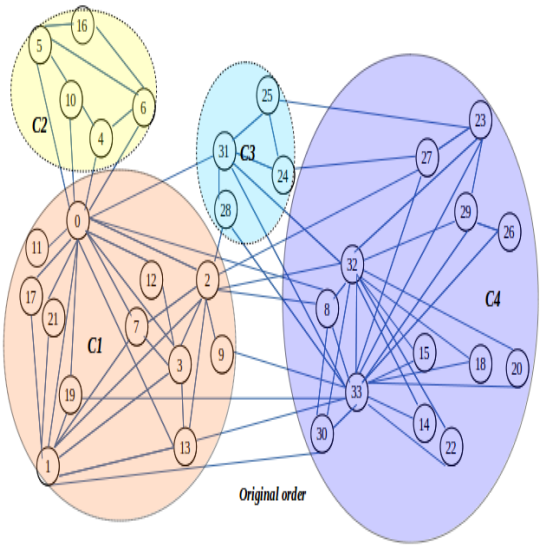

Figure 16: karate graph

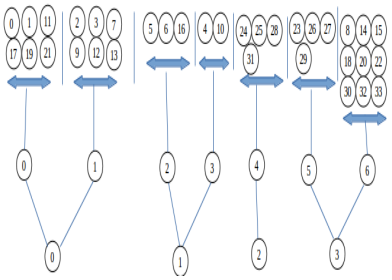

Figure 17: Hierarchy with Louvain

C Cache references reduction

Figure 19 shows cache references with different orders.

Figure 20 shows cache references with different orders.

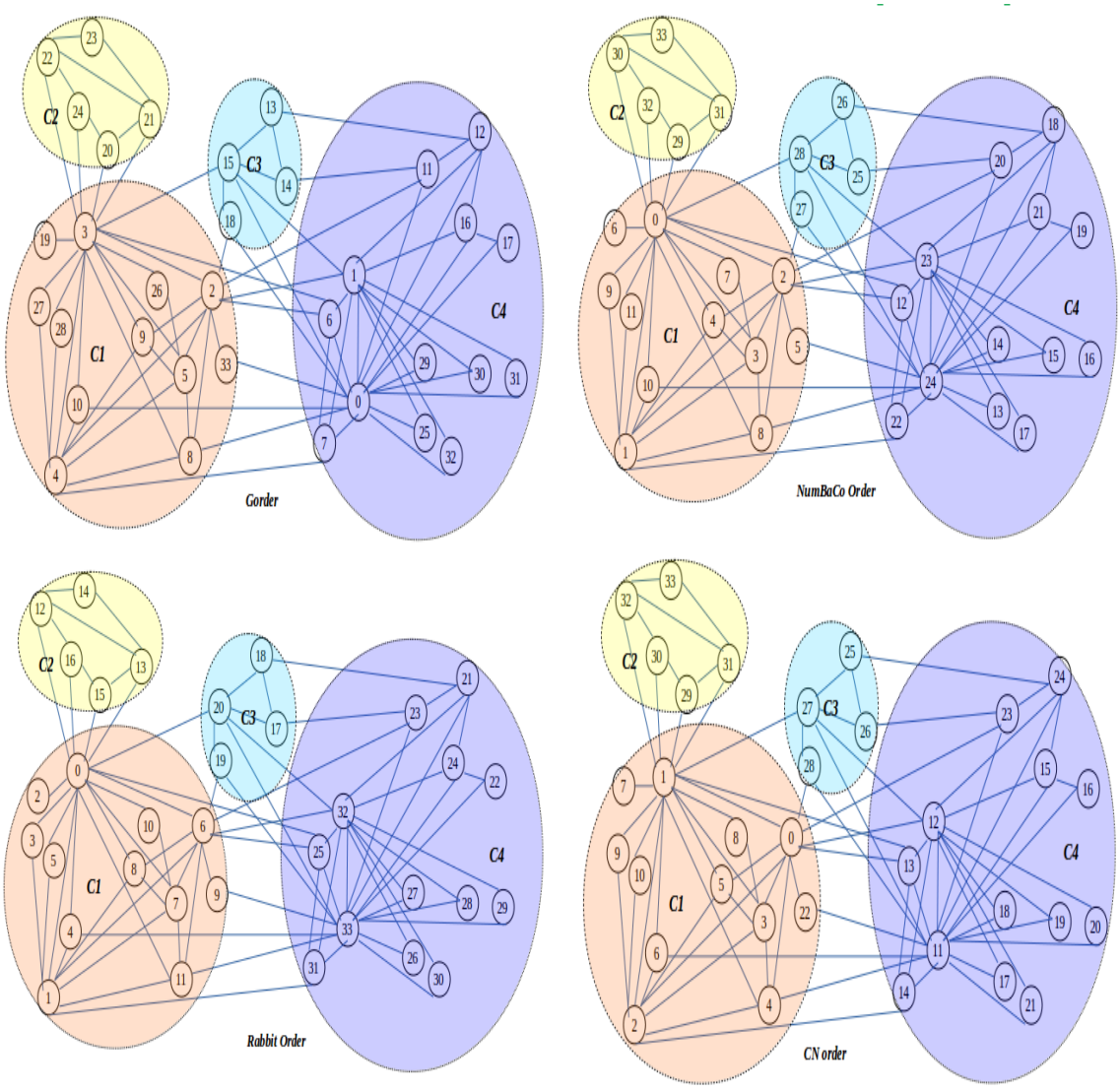

Figure 18: karate graph with different orders

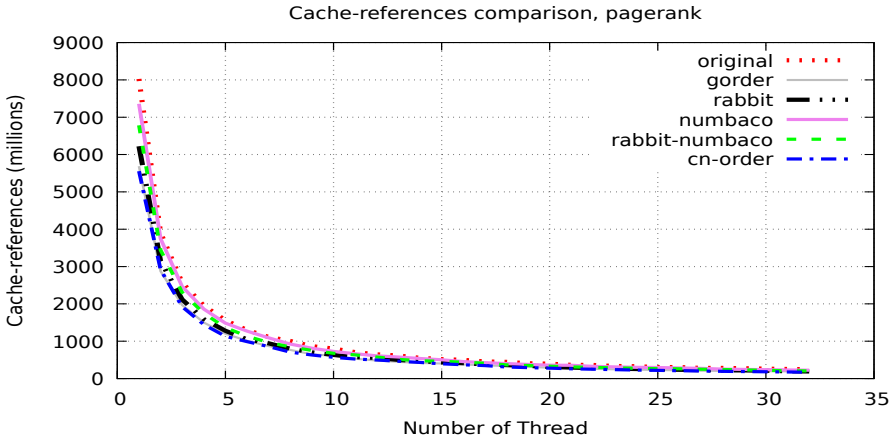

Figure 19: Cache references comparison with graph orders

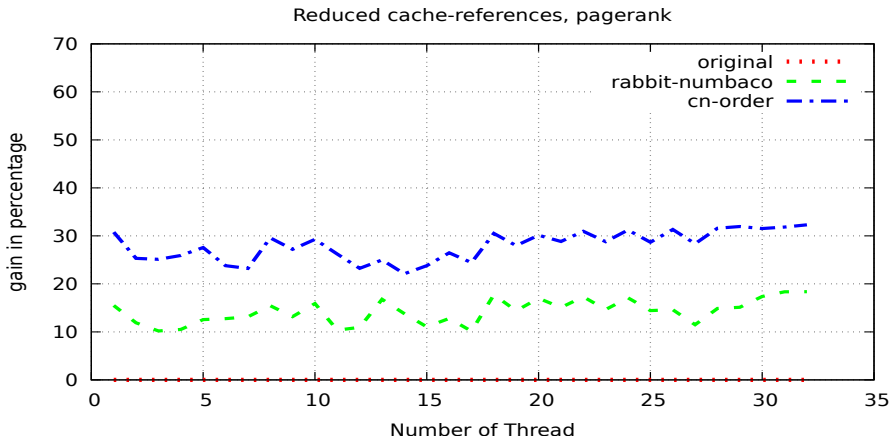

Figure 20: Reduced cache references with cn-order vs rabbit-numbaco