Using Complex-Network properties For Efficient Graph Analysis

Thomas Messi Nguélé, Maurice Tchuente, Jean-François Méhaut

▶ To cite this version:

Thomas Messi Nguélé, Maurice Tchuente, Jean-François Méhaut. Using Complex-Network properties For Efficient Graph Analysis. International Conference on Parallel Computing, ParCo 2017, Foundation ParCo Conferences and Consortium Cineca, Sep 2017, Bologne, Italy. pp.413 - 422, 10.3233/978-1-61499-843-3-413 . hal-01498578v2

HAL Id: hal-01498578
https://hal.science/hal-01498578v2

Submitted on 17 Nov 2017

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

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

Using Complex-Network Properties For Efficient Graph Analysis

Thomas MESSI NGUÉLÉ [a,b], Maurice TCHUENTE [b] and Jean-François MÉHAUT [b]
[a] *IRD, UMI 209 UMMISCO, Université de Yaoundé I, BP 337 Yaoundé Cameroun*
[b] *Université de Grenoble Alpes, INRIA-LIG, Corse, BP: 38400 Grenoble, France*

Abstract. A complex network is a set of entities in a relationship, modeled by a graph 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, because most well studied (parallel) abstractions and implementation are not suitable for them. This work shows how to use some complex-network properties, including community structure and heterogeneity of node degree, to tackle one of the main challenges in graph analysis applications: improving performance, by a proper memory management and an appropriate thread scheduling. In this paper, we first proposed Cn-order, a heuristic that combines advantages of the most recent algorithms (Gorder, Rabbit and NumBaCo) to reduce cache misses in graph algorithms. Second, we proposed deg-scheduling, a degree-aware scheduling to ensure load balancing in parallel graph applications. Then we proposed comm-deg-scheduling, an improved version of deg-scheduling that uses Cn-order to take into account graph order in scheduling. Experimental results on a 32 cores NUMA machine (NUMA4) (with Pagerank and livejournal for example) showed that Cn-order used with deg-scheduling (comm-deg-scheduling) outperforms the recent orders: with 32 threads, we reduce time by 26.81% compared to Gorder, 17.28% compared to Numbaco and 11.53% compared to Rabbit.

Keywords. graph analysis, performance, cache miss, scheduling, load balancing

1. Introduction

A network is a set of entities such as individuals or organizations, with connections between them. Such a system is modeled as a graph G = (V, E) where entities are represented by vertices in V, and connections are represented by edges in E. As outlined in [11], the advent of computers and communication networks that allow to gather and analyze data on a large scale, 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.

This huge size combined with the complexity of the questions raised, make the design of efficient algorithms for parallel complex network analysis a real challenge [9]. 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 [15].

On the other hand, the irregular structure of complex networks combined with the spatial locality of graph exploration algorithms, make difficult the scheduling task. In-

deed, 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 [17,18].

This paper addresses one of the major challenges of graph analysis [9]: improving performance through memory locality and thread scheduling. We did it with a proper storage of complex network in memory and by ensuring that workload among threads is well balanced.

Contribution. Despite the local irregularity of graph structures, it is known that complex networks have some macroscopic structures such as communities, i.e. groups of vertices that have a high density of edges within them and a lower density of edges between groups [11]. Our contribution is as follows:

1. In the first aspect of our contribution, we exploit community structure in order to design a node ordering that induces locality at a higher level during network exploration, yielding a reduction of cache misses. More precisely, we present Cn-order a new graph order heuristic that combines advantages of the most recent graph ordering heuristics. This advantages include bring closer in memory pairs of nodes appearing frequently in direct neighborhood (Gorder), or bring closer in memory nodes belonging to the same community or sub-community (NumBaCo, Rabbit).
2. In the second aspect, we introduce a technique that takes into account the heterogeneity of nodes degree to tackle load balancing among threads. This leads to a heuristic named deg-scheduling that, used with Cn-order takes into account the community and the heterogeneity of nodes degrees in order to obtain proper storage and balanced workload among threads. Comm-deg-scheduling (Cn-order used with deg-scheduling) outperforms the recent orders: on NUMA4, we reduce time by 26.81% compared to Gorder, 17.28% compared to Numbaco and 11.53% compared to Rabbit.

Paper organization. The remainder of this paper is structured as follows. Section 2 presents prerequisites such as complex networks representation, cache management and Common pattern in graph analysis. Section 3 presents the three most recent heuristics for graph ordering. Section 4 introduces our main contribution: section 4.1 presents a new graph ordering for efficient cache management during network analysis and, section 4.2 precises how to benefit to our heuristic according to the target graph algorithm complexity, section 4.3 is devoted to degree-aware loop scheduling that ensures load balancing among threads during parallel complex network analysis. Experimental results on algorithms that combine community-aware node ordering and degree-aware thread scheduling are presented in section 5. Section 6 is devoted to the synthesis of our contribution, together with the conclusion and future work.

2. Background

Complex network representation. There are three main ways to represent complex networks:

- *Matrix representation.* A simple way to represent this 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.

- *Yale representation.* The representation often adopted by some graph specific languages such as Galois [12] and Green-Marl [6], is that of Yale [5]. This representation uses three vectors: – vector A represents edges, each entry contains the weight of each existing edge (weights with same origin are consecutive), – vector JA gives one extremity of each edge of A, – and vector IA gives the index in vector A of each node (index in A of first stored edge that have this node as origin).

- *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 [14], - a linked list of blocks (with fixed size) of its neighbors, adapted to the dynamic graphs, used by the Stinger platform in [4,2].

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

```
1: for u ∈ V(G) do
2:     for v ∈ 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 as far as possible, that successive u and v are close in memory.

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.

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 will show how complex graphs can be organized to reduce cache misses.

3. Releated work: graph ordering heuristics

In this section, we present three recent algorithms: Gorder [19], Rabbit Order [1] and NumBaCo [10]; These three algorithms were published quite at the same period; each of them claims to outperform state of the art algorithms such as BFS order [7], graph partition order with METIS [8].

3.1. Gorder

The goal is to reduce cache misses by making sibling nodes be close in memory (allow them to fit in cache memory). 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 order to minimize the cpu cache miss ratio. Hao Wei and co-authors [19] defined a score function $S(u,v) = S_s(u,v) + S_n(u,v)$, where $S_s(u,v)$ is the number of common

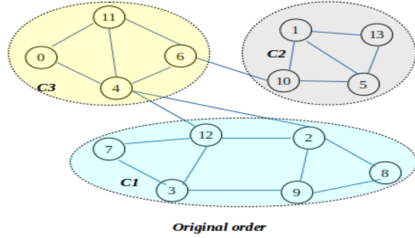

Figure 1. Running example

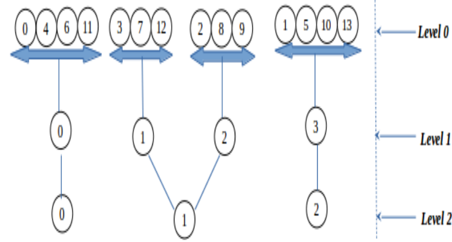

Figure 2. Running graph with Louvain

in-neighbors of u and v (sibling relationship); $S_n(u,v)$ is the number of times that u and v are in neighbor relationship. Based on this score function , Gorder algorithm tries to find a permutation π that maximize the sum of score $S(.,.)$:

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

3.2. NumBaCo and Rabbit Orders

The main idea of these orders is to make nodes that belong to the same community to close in memory. Both NumBaCo and Rabbit orders start by detecting communities in a graph before numbering. If NumBaCo uses Louvain algorithm [3], Rabbit order uses another community detection algorithm [16] that can be seen as a light version of Louvain.

Louvain algorithm [3] 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.

In contrast to Louvain, algorithm used in Rabbit [16] does not traverse all vertices multiple times; it incrementally aggregates vertices placed in a same community into an equivalent single, this contributes to reduce drastically execution time. To ensure just in time graph ordering, Junya Arai and co-authors [1] propose a parallel version of this community detection algorithm.

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 1, is given at figure 2. Level one has 4 communities {0,1,2,3}, level two (the last) has 3 communities {0,1,2}.

After detecting communities, if Rabbit generates directly a numbering, NumBaCo first classifies communities to make ones with higher affinity (number of edge shared) being together. The numbering is different in the two algorithms. Rabbit assigns a new number to each node by performing DFS (Depth-First Search) within each hierarchical community. This action makes nodes belong to the same sub-community be close in the memory. NumBaCo assigns numbers to nodes by keeping the initial order within each community: that is, if x and y are in the same community and $x < y$ then numbaco will assign $\pi(x)$ to x and $\pi(y)$ to y in such away that $\pi(x) < \pi(y)$.

3.3. Strengths and weaknesses of each heuristic

Advantage and disadvantage of gorder: 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 take into account the community structure which is usually present in complex networks and influences graph analysis performances. During the numbering, nodes that belong to same community may be scattered.

Advantage and disadvantage of NumBaCo and rabbit: 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. May be, a good numbering within communities can allow better performances.

The next paragraph shows how we use these strengths and weaknesses to build a more powerful graph order.

4. Community-aware graph ordering and degree-aware scheduling

The first part of this section presents a new graph ordering for efficient cache management based on community structure of complex-networks. In the second part, we present time complexities of all graph ordering heuristics, in order to categorize target graph algorithms. The last part present degree-aware loop scheduling that ensures load balancing among threads during parallel complex network analysis.

4.1. Community-aware graph ordering

The problem of complex-network ordering for cache misses reduction can be easily formalized as the optimal linear arrangement problem (a well known NP-Complete problem)[10]. As usual in this kind of problem, the only way to solve it in reasonable time is through heuristics. The goal of Cn-order (complex network order) heuristic is to bring close nodes of the same community with NumBaCo and within each community, use Gorder to bring close nodes appearing in a direct neighborhood. A simple approach of Cn-order is presented in Algorithm 1.

Algorithm 1 : **C**omplex **N**etwork order (**cn-order**)

Input: $G = (V,E)$
Output: $G' = \pi(G)$, π permutation, neighborhood storage is changed

1: $\pi_1 \leftarrow produce_\pi_1_in_goder_fashion(G)$
2: $Com \leftarrow detect_comm_Louvain(\pi_1(G))$
3: $Com_{cl} \leftarrow classify_and_find_offsets(Com, Cache_size)$
4: $\pi \leftarrow graph_numbering(Com_{cl}, \pi_1(G))$
5: $G' \leftarrow store_neighboors(G, \pi)$

Cn-order version 1. — Line 1 computes π_1 in Gorder fashion. Details are close to algorithm 4 with the only difference that, here it is applied to the whole graph.

Lines 2 to 5 compute NumBaCo order with little changes (at line 3 when cache memory size is taking into account): — line 2 computes communities with Louvain

algorithm [3]; — Line 3 classifies communities according to their affinities (algorithm 2, line 1 to 6); it also delimits communities in different groups in such away that each group will fit into cache memory with the size *Cache_size* (algorithm 2, line 7 to 20).

Algorithm 2 : $classify_and_find_offsets$

Input: $Com, Cache_size$
Output: Com_{class} communities ranged by their affinity, having each at most $Cache_size$ nodes

1: select $C \in Com$; $Com_{cl}[1] \leftarrow C$; $Com_r \leftarrow Com - \{C\}$
2: $i \leftarrow 2$, $nb_com \leftarrow Com.nb_com$
3: **while** $i \leq nb_com$ **do**
4: $\quad C_{max} \leftarrow arg \max_{\forall Com_r[k]} |\{(u,v) : u \in Com_{cl}[i-1], v \in com_r[k]\}|$
5: $\quad Com_{cl}[i] \leftarrow C_{max}$; $Com_r \leftarrow Com_r - \{C\}$; $i \leftarrow i+1$
6: **end while**
7: $i \leftarrow 1$; $offset \leftarrow 0$
8: **for all** $C \in Com_{cl}$ **do**
9: $\quad$ **if** $C.nb_edge \leq Cache_size$ **then**
10: $\quad\quad Com_{class}[i] \leftarrow C$,
11: $\quad\quad Com_{class}[i].offset \leftarrow offset$,
12: $\quad\quad i \leftarrow i+1$; $offset \leftarrow offset + C.nb_node$
13: $\quad$ **else**
14: $\quad\quad$ **for all** *subcommunity* $s_C \in Com_{cl}$ **do**
15: $\quad\quad\quad Com_{class}[i] \leftarrow s_C$,
16: $\quad\quad\quad Com_{class}[i].offset \leftarrow offset$,
17: $\quad\quad\quad i \leftarrow i+1$; $offset \leftarrow offset + s_C.nb_node$
18: $\quad\quad$ **end for**
19: $\quad$ **end if**
20: **end for**

— line 4 generates ordering by ensuring that nodes belonging to the same community have consecutive numbers and within each community, nodes keep gorder numbering

— Line 5 changes the storage of each node. It sorts the neighborhood of each node.

This version of cn-order reaches its initial goal : – sibling nodes are kept close, – nodes that belong to same community are close, – and communities with higher affinity are close. Another advantage of cn-order is that, it generates an order that takes into account the target architecture through cache memory size (thanks to line 3). The main disadvantage of this version of cn-order is in its time complexity: gorder time complexity + numbaco time complexity. To overcome this problem, we design another version of cn-order based on the same idea (Algorithm 3).

Cn-order version 2. The goal here is to reduce execution time and keeping the same ordering quality. The algorithm starts by communities detection at line 1. In this version of cn-order, we adopted communities detection algorithm proposed in [16] and precisely a a parallel communities detection based on this algorithm proposed in [1].

Line 5 of cn-order (detailed in Algorithm 4) generated order for each community. We adapted Gorder [19] to be applied in each community (each community is seen as a small graph).

Finally, to ensure that cn-order will take less execution time compared to first version of cn-order, we introduce parallelism (parallel community detection, computing order within communities in parallel, and parallel change neighborhood storage).

4.2. Time complexity comparison of graph ordering heuristics

Consider graph $G = (N, E)$, where $n = |N|$ and $m = |E|$. Table 1 presents time complexities of all heuristics. Among the first three, Gorder is the slowest. Rabbit and NumBaCo

Algorithm 3 :**C**omplex **N**etwork order (**cn-order_2**)

Input: $G = (V,E), Cache_size$
Output:$G' = \pi(G)$, π permutation, neighborhood storage is changed
1: $Com \leftarrow detect_communities(G)$
2: $Com_{cl} \leftarrow classify_and_find_offsets(Com, Cache_size)$
3: $nb_{class} \leftarrow Com_{cl}.nb_{class}$
4: **for all** $i \in [1 \dots nb_{class}]$ **parallel do**
5: $\quad compute_community_order(Com_{cl}[i], G, \pi)$
6: **end for**
7: $G' \leftarrow store_neighboors(G, \pi)$

Algorithm 4 : *compute_community_order* (Gorder [19] adapted to a community)

Input: $Com_{cl}, \pi, G, w, S(.,.)$: π will be set only for nodes in Com_{cl}
Output: π a permutation, for nodes in Com_{cl}
1: select $v \in Com_{cl}$, $o_set \leftarrow Com_{cl}.offset$
2: $\pi[1 + o_set] \leftarrow v$, $V_r \leftarrow \{x : x \in Com_{cl}\} - \{v\}$
3: $i \leftarrow 2$, $nb_nod \leftarrow Com_{cl}.nb_nod$
4: **while** $i \leq nb_nod$ **do**
5: $\quad v_{max} \leftarrow arg \max_{v \in V_r} \sum_{j=max\{1,i-w\}}^{i-1} S(P[j], v)$
6: $\quad \pi[i + o_set] \leftarrow v_{max}$
7: $\quad V_r \leftarrow V_r - \{v_{max}\}$, $i \leftarrow i + 1$
8: **end while**

Table 1. Time complexity comparison of all heuristics

Heuristic	Gorder	Rabbit	NumBaCo	Cn-order
Time complexity	$O(\sum_{v \in N} d_v^2 + n)$	$O(E - n)$	$O(nlogn)$	$O(\sum_{v \in N} d_v^2 + n(1 + logn))$

time complexities are due to their communities detection algorithm respectively. Since Cn-order uses Gorder and NumBaCo, it becomes the slowest one. In order to reduce Cn-order complexity, one should reduce Gorder and NumBaCo complexities. This can be done by increasing parallelization withing heuristics. It is for example what we did in the second version of Cn-order.

To benefit to all of these heuristics, 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 performance improvement due to heuristics, because the time to re-order the graph is negligible compared to the time the target graph algorithm. But for graph algorithms in **category 2**, to benefit to one of these heuristics (particularly Cn-order), one should used it for preprocessing.

4.3. *Degree-aware scheduling for load balancing*

Degree-aware scheduling problem can be formalized as multiple knapsack problem (a well known NP-complete problem). That is, the only way to solve it in a reasonable time is through a heuristic. The goal of the following heuristic is to find the workload of each thread. This workload is based on nodes degree. 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}. The complexity of this algorithm is O(n).

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: $\quad task[t].start \leftarrow i,\ d_{tmp} \leftarrow 0$
4: $\quad$ **while** $d_{tmp} < d_{max}$ **and** $i < n$ **do**
5: $\quad\quad d_{tmp} \leftarrow d_{tmp} + d_i;\ i \leftarrow i+1$
6: $\quad$ **end while**
7: $\quad task[t].deg \leftarrow d_{tmp},\ task[t].end \leftarrow i+1;\ i \leftarrow i+1$
8: **end while**

4.4. Graph ordering and scheduling for efficient graph analysis

We saw at section 4.1 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 algorithm 6. *Comm-deg-scheduling* first processes cn-order before scheduling. By that, it ensures that nodes processed by each thread will be closed in memory.

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($\pi(G), d_{sum}, nb_{thread}$)

5. Experimental evaluation

For this evaluation, we present results got with the well known graph analysis application, Pagerank [13]. We used a posix thread implementation proposed by *Nikos Katirtzis*[1]. This implementation uses adjacency list representation. The experiments were made on a machine (NUMA4) 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.

5.1. Graph ordering comparison

We did this experimentation on NUMA4 described above; the dataset is livejournal [20] unoriented graph with 3997962 nodes and 34681189(X2) 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).

Results in table 2 (with one thread) confirm the strengths and weaknesses of each algorithm presented in section 3.3: — 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 that Gorder brings sibling nodes close compared to the others; — NumBaCo has the least cache misses, close to Rabbit order. Numbaco does better than Rabbit because after detecting communities, it also classifies them according to their affinities. — CN-order (simple version) outperforms all the orders, since it combines all the advantages: 33.38% of cache references, 45.02% of cache misses and 29.60% of execution time.

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

Table 2. Performances comparison (Pagerank, 1 and 32 threads)

Heuristic		Original	Gorder	Rabbit	NumBaCo	**cn-order**	**com-deg-sched**
L3 cache	1	7,689	5,686 (**26.05%**)	6,201 (19.35%)	5,888 (23.42%)	5,122 (**33.38%**)	5,129 (33.29%)
ref(M)	32	8,277	6,028 (**27.17%**)	6,907 (16.55%)	6,465 (21.89%)	5,454 (34.10%)	5,402 (**34.73%**)
L3 cache	1	3,356	2,997 (10.69%)	2,153 (35.84%)	1,845 (**45.02%**)	1,809 (**46.09%**)	1,811 (46.03%)
miss(M)	32	4,212	3,342 (20.65%)	3,359 (20.25%)	2,914 (**30.81%**)	2,591 (**38.48%**)	2,736 (35.04%)
Time (s)	1	345.297	310.851 (09.97%)	273.832 (20.69%)	250.313 (**27.56%**)	243.070 (29.60%)	242.997 (**29.62%**)
	32	71.105	53.843 (24.27%)	42.981 (**39.55%**)	47.071 (33.80%)	44.638 (37.22%)	34.779 (**51.08%**)

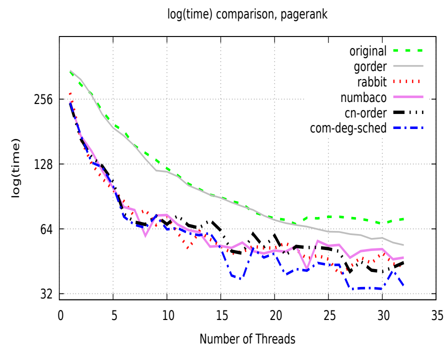

Figure 3. Time comparison with graph orders

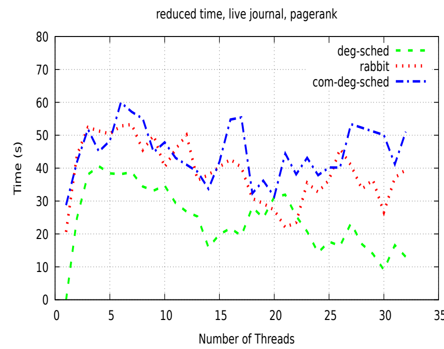

Figure 4. Performance due to scheduling & ordering

Observations done with one thread are quite the same with multiple threads (from 2 to 32) in terms of cache misses reduction and cache references reduction. It is different with execution time reduction. In figure 3, cn-order, Rabbit, Numbaco curves are switching their positions; this is explained by the imbalance load that arises among threads: according to the number of threads, the curve below the others corresponds to the one which is associate to the best load balancing among threads. This is the reason why, comm-deg-scheduling (which ensures load balancing among threads) is almost belong the others (cn-order, Numbaco, Rabbit, Gorder).

5.2. Graph ordering and degree-aware scheduling

Figure 4 presents the reduced time (in percentage) due to degree-aware scheduling. Consider time obtained with 32 threads: — compared to node-aware scheduling, when using degree-aware scheduling, time is reduced by 13.04%; — compared to node-aware scheduling, using comm-deg-scheduling allows to reduce time from 37.22% (with Cn-order) to 51.08% (with Cn-order and deg-scheduling). This figure also show that it outperforms Rabbit by 11.53%.

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.

6. Conclusion

In this paper, we first propose Cn-order, a heuristic that combines advantages of the most recent algorithms (Gorder, Rabbit and NumBaCo) to solve the problem of complex-network ordering for cache misses reduction, a problem formalized as the optimal linear arrangement problem (a well known NP-Complete problem).

Second, we proposed deg-scheduling, a heuristic to solve degree-aware scheduling problem, a problem formalized as multiple knapsack problem (also known as

NP-complete). Then we proposed comm-deg-scheduling, an improved version of deg-scheduling that uses Cn-order to take into account graph order in scheduling.

Experimental results on a 32 cores NUMA machine (with Pagerank and livejournal for example) showed that Cn-order used with deg-scheduling (comm-deg-scheduling) outperforms the recent orders: with 32 threads, we reduce time by 26.81% compared to Gorder, 17.28% compared to Numbaco and 11.53% compared to Rabbit.

As future work, we plan to implement these heuristics in graph analysis platforms such as Galois [12] or Green-Marl [6] in order to improve their performances.

References

[1] 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.

[2] 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.

[3] V. Blondel, J-L Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

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

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

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

[7] K. Karantasis, A. Lenharth, D. Nguyen, M. Garzarán, and K. Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the IC HPC, Networking, Storage and Analysis*, pages 921–932. IEEE Press, 2014.

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

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

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

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

[12] 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.

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

[14] Jason Riedy, David A. Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. *IEEE Computer Society Washington, DC, USA 2012*, 18(1):1619–1628, 2012.

[15] M. Rosenblum, E. Bugnion, S. Alan Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. *SIGOPS Operating Systems Review*, 29(5):285–298, 1995.

[16] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. Fast algorithm for modularity-based graph clustering. In *AAAI*, pages 1170–1176, 2013.

[17] Fengguang Song, Shirley Moore, and Jack Dongarra. Feedback-directed thread scheduling with memory considerations. In *16th international symposium on HPDC*, pages 97–106. ACM, 2007.

[18] Fengguang Song, Shirley Moore, and Jack Dongarra. Analytical modeling for affinity-based thread scheduling on multicore platforms. *Symposim on Principles and PPP*, 2009.

[19] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 ICMD*, pages 1813–1828. ACM, 2016.

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