DSL pour la fouille des réseaux sociaux sur des architectures Multi-coeurs

Thomas Messi Nguele

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

et de :

DOCTEUR (Ph.D) DE L'UNIVERSITE de YAOUNDE 1

préparée dans le cadre d'une cotutelle *entre la Communauté Université Grenoble Alpes* et l'Université de Yaoundé 1

Spécialité : **Informatique**

Arrêté ministériel : le 25 mai 2016 (France)
le 23 septembre 1999 - 11 avril 2001 (Cameroun)

Présentée par

MESSI NGUELE Thomas

Thèse codirigée par
Maurice TCHUENTE et **Jean-François MÉHAUT**

préparée au sein des **Laboratoires LIG** et **UMMISCO**
dans **les Écoles Doctorales ED MSTII et CRFD STG**

DSL pour la Fouille des Réseaux Sociaux sur les Architectures Multi-coeurs

DSL for Social Networks Analysis on Multi-core Architectures

Thèse soutenue publiquement le **« 15/09/2018 »**,
devant le jury composé de :

Pr. Roger ATSA ETOUNDI
Professeur Université de Yaoundé 1, Président
Pr. Clémentin TAYOU
Maitre de Conférence Université de Dschang, Rapporteur
Pr. Emmanuel VIENNET
Professeur Université Paris Nord, Rapporteur
Pr. Daniel AKUME
Maitre de Conférence Université de Buéa, Examinateur
Pr. Maurice TCHUENTE
Professeur Université de Yaoundé 1, Co-directeur de thèse
Pr. Jean-François MÉHAUT
Professeur Université de Grenoble Alpes, Co-directeur de thèse

Domain Specific Languages For Social Networks Analysis on Multi-core Architectures

Thomas Messi Nguélé
Co-supervisor (UY1): Pr Maurice Tchuente
Co-supervisor (UGA): Pr Jean-Francois Méhaut

September 25, 2018

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. The main question in this thesis is how to develop graph analysis applications that are both –easy to write (implementation challenge), – and efficient (performance challenge)? We answer this question with parallelism (parallel DSLs) and also with knowledge that we have on complex networks (complex networks properties such as community structure and heterogeneity of node degree).

The first contribution of this thesis shows the exploitation of community structure in order to design community-aware graph ordering for cache misses reduction. We proposed NumBaCo and compared it with Gorder and Rabbit (which appeared in the literature at the same period NumBaCo was proposed). This comparison allowed to design Cn-order, another heuristic that combines advantages of the three algorithms (Gorder, Rabbit and NumBaCo) to solve the problem of complex-network ordering for cache misses reduction. Experimental results with one thread on Core2, Numa4 and Numa24 (with Pagerank and livejournal for example) showed that Cn-order uses well the advantages of the other orders and outperforms them.

The second contribution of this thesis considered the case of multiple threads applications. In that case, cache misses reduction was not sufficient to ensure execution time reduction; one should also take into account load balancing among threads. In that way, heterogeneity of node degree was used in order to design Deg-scheduling, a heuristic to solve degree-aware scheduling problem. Deg-scheduling was combined to Cn-order, NumBaCo, Rabbit, and Gorder to form respectively Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling. Experimental results with many threads on Numa4 showed that Degree-aware scheduling heuristics (Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling) outperform their homologous graph ordering heuristics (Cn-order, NumBaCo, Rabbit, and Gorder) when they are compared two by two.

The last contribution was the integration of graph ordering heuristics and degree-aware scheduling heuristics in graph DSLs and particularly Galois and Green-Marl DSLs. We showed that with Green-Marl, performances are increased by both graph ordering heuristics and degree-aware scheduling heuristics (time was reduced by 35% due to heuristics). But with Galois, performances are increased only with graph ordering heuristics (time was reduced by 48% due to heuristics).

In perspective, instead of using complex networks properties to design heuristics, one can imagine to use machine learning. Another perspective concerns the theoretical aspect of this thesis. We showed that graph ordering for cache misses reduction and degree-aware scheduling for load balancing problems are NP-complete. We provided heuristics to solve them. But we didn't show how far these heuristics are to the optimal solutions. It is good to know it in the future.

Résumé

Les réseaux complexes sont des ensembles constitués d'un grand nombre d'entités interconnectées par des liens. Ils sont modélisés par des graphes dans lesquels les noeuds représentent les entités et les arêtes entre les noeuds représentent les liens entre ces entités. Ces graphes se caractérisent par un très grand nombre de sommets et une très faible densité de liens. Les réseaux sociaux sont des exemples de réseaux complexes où les entités sont des individus et les liens sont les relations (d'amitié, d'échange de messages) entre ces individus. Le travail réalisé dans cette thèse était lié au développement d'applications d'analyse des réseaux sociaux, qui soient à la fois faciles à écrire et efficaces. A cet effet, deux pistes ont été explorées: a)L'exploitation de la structure en communautés pour définir des techniques de stockage qui réduisent les défauts de cache lors de l'analyse des réseaux sociaux; b)La prise en compte de l'hétérogénéité des degrés des noeuds pour optimiser la mise en oeuvre parallèle.

La première contribution de cette thèse met en évidence l'exploitation de la structure en communautés des réseaux complexes pour la conception des algorithmes de numérotation des graphes (NumBaCo, CN-order) permettant la réduction des défauts de cache des applications tournant dans ces graphes.Les résultats expérimentaux en mode séquentiel sur plusieurs architectures (comme Numa4) ont montré que les défauts de cache et ensuite le temps d'exécution étaient effectivement réduits; et que CN-order se sert bien des avantages des autres heuristiques de numérotation (Gorder, Rabbit, NumBaCo) pour produire les meilleurs résultats.

La deuxième contribution de cette thèse a considéré le cas des applications multi-threadées. Dans ce cas, la réduction des défauts de cache n'est pas suffisante pour assurer la diminution du temps d'exécution; l'équilibre des charges entre les threads doit être assuré pour éviter que certains threads prennent du retard et ralentissent ainsi toute l'application. Dans ce sens, nous nous sommes servis de la propriéte de l'hétérogénéité des dégrés des noeuds pour développer l'heuristique Deg-scheduling. Les résultats expérimentaux avec plusieurs threads sur l'architecture Numa4 montrent que Deg-scheduling combiné aux heuristiques de numérotation permet d'obtenir de meilleur résultats.

La dernière contribution de cette thèse porte sur l'intégration des deux catégories d'heuristiques développées dans les DSLs parallèles d'analyse des graphes. Par exemple, avec le DSL Green-Marl, les performances sont améliorées à la fois grâce aux heuristiques de numérotation et grâce aux heuristiques d'ordonnancement (temps réduit de 35% grâce aux heuristiques). Mais avec le DSL Galois, les performances sont améliorées uniquement grâce aux heuristiques de numérotation (réduction de 48%).

En perspective, au lieu d'utiliser les propriétés des réseaux complexes pour développer les heuristiques, on peut imaginer l'usage des méthodes de machine learning. Une autre perspective concerne l'aspect théorique de la thèse. Nous avons montré que le problème de numérotation des graphes pour la diminution des défauts de cache et le problème de l'ordonnancement basé sur les dégrés des noeuds pour l'équlibrage des charges sont NP-complets. Nous avons proposé des heuristiques pour les résoudre, mais nous n'avons pas montrer à quelle distance ces heuristiques se situent de la solution optimale. Ceci reste une question ouverte qui mérite d'être étudiée.

To my mum Metili Mbang Marthe Félicité,
In memory of my dad Nguélé Oloa Simon,
In memory of my grandmother Tiga Marceline Thérèse.

Acknowledgements.

I thank all of you who contributed to bring the accomplishment of this thesis. First of all, I thank you my supervisors, Professors Jean-François Méhaut and Maurice Tchuente for the help and the time you consecrated to guide and accompany me through the years. You expressed your availability from the beginning to the end of this thesis and especially during difficult times. You two trusted me since I was in master 2. I hope that the outcome of this thesis gives you great satisfaction.

Thank you Professor Clémentin Tayou, Professor Emmanuel Viennet and Professor Daniel Akume for agreeing to participate in the jury of this thesis. More particularly Professor Emmanuel Viennet, I thank you for your investment in research at the University of Yaounde I. I hope this thesis will help to reinforce this investment.

I thank the laboratories LIRIMA, UMMISCO, LIG, INRIA who funded this thesis through: - the scholarship awarded by the doctoral school of Grenoble, - flight tickets, - and support during summer schools, team seminars or conferences.

A special thank you to members of the Corse team for the friendly atmosphere: the chief Fabrice Rastello (always full of energy) and permanent researchers Frédéric Desprez (always fashionable), Yliès Falcone (Vice Chief), François Broquedis (Mr. OpenMP), Florent Bouchez (Mr. Compilation), and Imma Presseguer, the team assistant, a special thanks to you for the gentleness and professionalism that made me forget the stress of the thesis work.

My heartfelt gratitude to you my teachers of University of Yaounde I (Professors Atsa, Ndoundam, Fotso, Doctors Tindo, Melatagia, Ngoko, Chedom, Moto, Tsopze, Soupgui, Kouokam, Nzali, Kamgnia and Tapamo) for your valuable advice and especially the quality of the training you gave me. Your efforts constituted an indispensable prerequisite to the completion of this thesis.

I thank my colleagues and friends from Yaoundé (Vanel Siyou, Robert Nsaibirni, Armel Nzekon, Samuel Tamene, Arielle Kitio, Kely Motue, Requi Djomo, Zenobie Tanekeng, Eric Mimbé, Leonie Tamo, Emmanuel Moupojou, Philippe Mah Tsila, Martial Juengue, Mariane Lisette Ndzana ...) and from Grenoble (Luis Felipe, Antoine El-Hokayem, Fabian Gruber, Raphaël Jakse, ...) for the many discussions sometimes beyond research but always rewarding.

Finally, I thank you, my family members: my brothers Ambroise Ondoa, Elie Mbarga, Arnaud Aloa, Simon Nguelé, Donatien Metili and my sisters Flore Ntolo, Hélène Elouma and Claire Ba'ana who were always there to encourage me.

Remerciements.

Je vous remercie, vous tous qui de près ou de loin avez contribué à mener à terme cette thèse. En premier lieu, je vous remercie vous mes directeurs de thèse, les professeurs Jean-François Méhaut et Maurice Tchuente pour l'aide et le temps que vous m'avez consacrés. Vous avez exprimé votre disponibilité du début à la fin de cette thèse et surtout pendant les périodes difficiles. Vous avez su me faire confiance dès le master 2. J'espère que l'aboutissement de cette thèse vous donne une grande satisfaction.

Je vous remercie Professeur Clémentin Tayou, Professeur Emmanuel Viennet et Professeur Daniel Akume pour avoir accepté de participer à ce jury de thèse. En particulier Professeur Emmanuel Viennet, je vous remercie en plus pour votre investissement dans la recherche à l'université de Yaoundé I. J'espère cette thèse contribuera à conforter cet investissement.

Je remercie les laboratoires LIRIMA-UMMISCO, LIG-INRIA qui ont financé cette thèse à travers – la bourse accordée par l'école doctorale de Grenoble, – les billets d'avion, – et les prises en charge lors des écoles d'été, des séminaires d'équipe ou des conférences.

Je vous remercie, vous les membres de l'équipe corse pour l'ambiance conviviale: le Chef Fabrice Rastello (toujours plein d'énergie) et les permanents Frédéric Desprez (toujours à la mode), Yliès Falcone (le vice Chef), François Broquedis (Monsieur OpenMP), Florent Bouchez (Monsieur Compilation), et Imma Presseguer, l'assistante d'équipe, je te remercie pour la douceur et le professionnalisme qui faisaient oublier le stress du travail de thèse.

Je vous remercie, vous mes enseignants de Yaoundé I (les professeurs Atsa, Ndoundam, Fotso, les docteurs Tindo, Melatagia, Ngoko, Chedom, Moto, Tsopze, Soupgui, Kouokam, Nzali, Kamgnia et Tapamo) pour vos précieux conseils et surtout la qualité de la formation dont vous m'avez fait bénéficier et qui a contribué à l'aboutissement de cette thèse.

Je vous remercie, vous mes collègues et amis de Yaoundé (Vanel Siyou, Robert Nsaibirni, Armel Nzekon, Samuel Tamene, Arielle Kitio, Kely Motue, Requis Djomo, Zénobie Tanekeng, Eric Mimbé, Léonie Tamo, Emmanuel Moupojou, Philippe Mah Tsila, Martial Juengue, Mariane Lisette Ndzana ...) et de Grenoble (Luis Felipe, Antoine El-Hokayem, Fabian Gruber, Raphaël Jakse, ...) pour les nombreuses discussions parfois au delà de la recherche mais toujours enrichissantes.

Je vous remercie enfin, vous les membres de ma famille: mes frères Ambroise Ondoa, Élie Mbarga, Arnaud Aloa, Simon Nguelé, Donatien Metili et mes soeurs Flore Ntolo, Hélène Elouma, Claire Ba'ana qui avez toujours su m'encourager.

Contents

Introduction

Chapter 1

Introduction

A complex network is a set made of a large number of entities interconnected with links. Social networks are an example of complex networks where entities are individuals and links are relationship (friendship, message passing or other) between these individuals. Complex networks are modeled by graphs where nodes represent entities and edges between nodes represent links between entities. Graphs that represent complex networks are usually big with sometime millions of nodes and millions or even billions of edges. For example at the end of the year 2017, Facebook social network boasts 2.13 billion active users per month and Twitter social network boasts 330 millions active users per month. This represents large volumes of data and leads some graph analysis applications to have long execution times. In order to have an idea of the impact of graph size in the execution time of a graph analysis application, let take Katz score application running on four datasets: Karate (a social network of members of a karate club in United States), Quant-ph (a social network of researchers in quantum physic), DBLP (a social network of scientific article authors in computer science) and the social network Live Journal.

Katz score can be used for links prediction of a given graph. For a non-oriented graph with n nodes and m edges, the number of links to predict is $\frac{n(n-1)}{2} - 2m$. Let t be the time necessary to predict a link between two nodes with Katz score. Naively, i.e without any strategy to reduce time, the execution time taken by Katz score application is given by $T = (\frac{n(n-1)}{2} - 2m)t$. With Karate graph which has 35 nodes and 79 links $T_{karate} = (\frac{35*34}{2} - 2*79)t = 516t$. With Quant-ph graph which has 1060 nodes and 1044 edges, $T_{quant-ph} = 5.59 * 10^5 t$. With DBLP graph which has 195310 nodes and 2099732 edges, $T_{dblp} = 1.9*10^{10}t$. With Live Journal graph which has 3997962 nodes and 34681189 edges, $T_{liveJ} = 7.99 * 10^{12}t$.

In [34], the best implementation of Katz score algorithm with Karate dataset took

$10.496s$. So an approximated value of t for Karate dataset is $t = \frac{T_{karate}}{516} = 2.03 * 10^{-2}s$. We assume that t has the same value for all the datasets. — But in practice, it is not really the case. Indeed, t value depends also of the graph size that influences cache misses of the application and hence the execution time. — Therefore, with $t = 2.03 * 10^{-2}s$:

$$\begin{cases} T_{karate} & = 10.496 \text{ seconds} \\ T_{quant-ph} & = 1.13 * 10^4 \text{ seconds} = 3.16 \text{ hours} \\ T_{dblp} & = 1.07 * 10^5 \text{ hours} = 12.29 \text{ years} \\ T_{liveJ} & = 5.15 * 10^3 \text{ years} \end{cases}$$

This example shows that, without taking into account the size of graphs in applications coming from complex networks analysis, the execution time can be very long (many hours or even years of execution if nothing is done). One way to reduce the execution time is using approximate solutions: rather than finding an exact calculation, some authors look for an approximate calculation that takes less time. It is the case with Paolo *et al.* [50] who choose a limited number of graph nodes to calculate an approximate value of *Betweenness Centrality*. In order to have an exact solution in a reasonable time, another one can use parallelism through high performance computers. However, the programmer may have difficulties to write efficient parallel programs that run on these architectures because he often finds himself writing low level platform specific code.

Before highlighting the goals and the contributions of this thesis, we will first present complex network properties, challenges in complex networks analysis and existing complex networks analysis parallelization approaches.

1.1 Complex Networks Properties

In addition to the big size that characterizes graphs, complex networks exhibit many other properties that distinguish them to random networks [32]. Some of these properties are described below:

- *Small world effect.* It describes the phenomenon that most pairs of nodes in many complex networks seem to be connected by a short path through the network. Stanley Milgram shows in the 1960s that letters passed from person to person were able to reach a designated target individual in only a small number of steps (around six steps).

- *Transitivity.* It is defined as follows: if node a is connected to node b and node b is connected to node c, then there is a heightened probability that node a will also be connected to node c. In social networks (like Facebook), it means the friend of your friend is likely to be your friend.

- *Heterogeneity of node degree.* The degree of a node in a graph is the number of nodes connected (with an edge) to that node. The heterogeneity of node degree is characterized by the fact that there are (usually a small number of) nodes with higher degree compared to other nodes with smaller degree. In social networks nodes with higher degree are famous nodes.

- *Community structure*: A graph has a community structure if it is characterized by groups of nodes that have a higher density of edges within them, and a lower density of edges between other groups.

We argue in this thesis that a proper exploitation of these properties coupled with a proper exploitation of the target architectures can allow to reduce execution time of graph analysis applications. In the next section we will remember challenges in complex networks analysis that if taken up will lead to reduce the execution time.

1.2 Challenges in Complex Networks Analysis

The study of graphs is not recent. It started in 1736 with the bridges of Königsberg problem [15]. But since the advent of the web a few decades ago, graph analysis gains more interest. One reason of this interest is the huge among of data generated by complex networks (in particular social networks). This huge among of data requires high performance computers to be analyzed in a reasonable time.

On the other hand, applications from complex networks analysis are generally based on the exploration of the underlying graph. This exploration is most often local: after treating a node, the next nodes to be referenced belong to the neighborhood of that node. Since the underlying graph is usually unstructured, data access patterns of this applications tend to have poor locality. In addition, complex networks analysis applications are often irregular, i.e, all the computations required by each node in the application are not usually well known *a priori*. In the case of parallel (multi-threaded) applications, this irregularity can lead to an unbalance load among the workers (threads).

All those characteristics of complex networks applications — i.e. large amount of data to analyze, irregularity of applications, poor locality — give rise to many challenges that

can be summarized in three mains: capacity, implementation and performance.

- **Capacity Challenge.** The capacity challenge refers to the fact that a dataset of a complex network application does not always fit into a single physical memory. In this thesis, we do not try to meet the capacity challenge. We consider the case where a dataset fits into a single physical memory, i.e simple memory machines, shared memory machines or NUMA (Non Uniform Memory Access) machines. We do not consider the case of distributed memory machines.

- **Implementation Challenge.** The implementation challenge is about writing easily a graph application from a given graph algorithm with a given programming language. Implementation challenge can be met with a programming language that is easy to use like Python or R programming languages. This challenge can also be met with a language that has its syntax close to graph analysis domain.

- **Performance Challenge.** In the performance challenge, we are looking for an efficient implementation of a given graph algorithm. An efficient implementation is the one that has the least execution time. This challenge is particularly difficult to met because of the irregularity and the poor locality of graph analysis applications. We think we can meet this challenge with an efficient parallelism ensured by a proper exploitation of complex network properties.

We have already mentioned complex networks properties in the previous section. Parallelism is not new in complex network analysis. In the next section, we will detail existing approaches in parallel complex networks analysis.

1.3 Parallelism in Complex Network Analysis

There are two main approaches to do parallelism in graph analysis: using programming models or using DSLs. In addition to know the syntax of the different languages used, the complex networks expert (complex networks analyzer) who chooses programming models should, master parallelization domain; i.e he should master all around parallelism such as Flynn's taxonomy [16, 12], models to use, the way to design parallel programs, etc. But the complex network analyzer who chooses a parallel DSL should only know the syntax of this DSL.

1.3.1 Using Programming Models

Starting with a reference algorithm (assumed to be sequential), there are three steps that lead to a parallel program.

> ***Step 1)*** After designing his new algorithm, a complex networks analyzer needs to perform some experimental validation. That is, he wants to know if the implementation of his algorithm in a programming language produces the right results for given inputs. In other words, the complex networks analyzer is looking for a programming language that allows him to make a prototype in order to validate his algorithm. For this purpose, he uses programming languages like Python or R, which are known for their ease of use.
>
> ***Step 2)*** After the prototyping step, the complex networks analyzer is now looking for a language that allows him to study performance issues of its algorithm. Programming languages like C or C++ are often used for this.
>
> ***Step 3)*** At this step, the complex networks analyzer wants to parallelize his program in order to get more performances. In this case, he uses one of the existing programming models such as CUDA, Posix threads, MPI or OpenMP. For example, community detection is implemented with OpenMP in [38] or with CUDA in [46].

It is not easy for the complex networks analyzer to follow all the three steps. This is because in addition to master the syntaxes of the used languages (Python or R, C or C++), it requires expertise in parallelism. There is another alternative that does not need so much efforts but that also ensures parallel graph analysis: parallel DSLs.

1.3.2 Using Parallel DSLs

A Domain-Specific Language (DSL) is defined as a language designed for a precise domain (for example SQL) as opposed to a general purpose language (for example Java, C). A DSL is either stand alone (has its own compiler) or embedded in an existing general purpose language.

Parallel DSLs in graph analysis are mobilizing more and more scientific community. For example, Hassan Chafi *et al.* [9] propose a generic DSL intended to be used by parallel DSL designers; Green-Marl [19] is a stand-alone DSL developed for parallel graph analysis; Galois [33] is another DSL designed for graph analysis, but it is an embedded one.

When using parallel graph DSLs, the only effort asked to a complex network analyzer is to learn the DSL syntax. All the efforts needed to get an efficient implementation are transparent to him.

1.4 Thesis Goal and Methodology

The general question that emerges is: how to develop graph analysis applications that are both *easy to write* (implementation challenge) and *efficient* (performance challenge)? We try to answer this question with parallelism and also with complex networks properties.

We saw in the previous section that we can do parallel graph analysis with programming models or with DSLs. While the first approach leads to efficient graph applications (performance challenge), the second targets both the performance and the implementation challenges. In this thesis, we choose the second approach. We are looking for a DSL that ensures easy and efficient complex networks analysis.

There are many DSLs designed for graph analysis. But all those DSLs were designed to implement classical graph algorithms. In that way, the implementation of the new algorithms arisen by complex networks with these DSLs is not always efficient. We argue in this thesis that one way to ensure this implementation to be efficient is by exploiting complex network properties. So the wanted DSL may exploit these properties in order to produce efficient implementation. Since none of the existing DSLs does not exploit these properties, we have two options: build a new DSL or use existing one. We choose the second option. Our methodology implies:

1. Exploit some complex networks properties in order to provide heuristics that improve graph applications performances.

2. Then, implement these heuristics in existing graph DSLs (Galois and Green-Marl).

The following section presents a reading guide of this thesis together with the contribution.

1.5 Contribution and Reading Guide

This thesis is divided into two parts. The first part proposes two categories of heuristics designed from complex networks properties. This part contains chapter 2, chapter 3 and chapter 4. The second part shows the implementation of the designed heuristics in graph DSLs. Chapter 5 and chapter 6 belong to that part.

1.5.1 Heuristics Designed For Efficient Graph Analysis

In this part, we exploit two complex networks properties to provide heuristics for efficient graph analysis. Precisely, we used *community structure* to design heuristics that reduce cache misses during an execution of a complex network analysis application. We also used *heterogeneity of node degree* to design heuristics that ensure load balancing among threads of a parallel complex network analysis application. The three chapters of this part are as follows.

Background. Chapter 2 gives the background necessary to understand chapter 3 and chapter 4: complex networks representation, graph analysis examples, multi-core architectures, cache-management, cache-references and cache-misses events (from Perf tool).

Community-aware Graph Ordering Heuristics. Chapter 3 presents the first contribution: graph ordering heuristics to ensure cache misses reduction. This chapter first proposes NumBaCo heuristic. It also presents Cn-order, another heuristic that takes into account characteristics of the target architecture and combines advantages of NumBaCo and two recent algorithms (Gorder, Rabbit) in order to solve the problem of complex-network ordering for cache misses reduction, a problem that we formalized as the optimal linear arrangement problem (a well known NP-Complete problem).

Degree-aware Scheduling Heuristics. Chapter 4 presents the second contribution. We used *heterogeneity of node degree* property to design scheduling heuristics that ensure load balancing among threads in parallel graph applications. We first proposes Deg-scheduling, a heuristic to solve degree-aware scheduling problem, a problem that we formalized as multiple knapsack problem (also known as NP-complete). Then it proposes Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling the improved version of Deg-scheduling that use respectively Cn-order, NumBaCo, Rabbit, and Gorder to take into account graph order in scheduling.

1.5.2 Heuristics Implementation in DSLs

The second part of this thesis is dedicated to the integration of designed heuristics in existing graph DSLs. The two chapters of this part are as follows.

State of The Art on Graph DSLs. Before presenting heuristics integration at chapter 6, chapter 5 recalls generalities about DSLs: - what is a DSL? - In which case to

develop a DSL? It also presents Graph DSLs with an emphasis in Galois and Green-Marl.

Heuristics Integration in Graph DSLs. Chapter 6 gives the third contribution of this thesis. It presents the implementation of graph ordering heuristics and degree-aware scheduling heuristics in Galois and Green-Marl, the two DSLs studied in chapter 5.

Conclusion. Chapter 7 concludes the thesis with a general assessment and general perspectives.

Part I

Heuristics For Efficient Graph Analysis

Chapter 2

Background

With the development of the web (search engines, e-commerce), complex networks are gaining more and more interest. We need high performance computers to efficiently process the graph applications that they induce. Using "only" parallel computers in graph applications is not sufficient to ensure efficiency. We show in this thesis how to combine parallelization with other strategies that increase efficiency.

Most algorithms from complex networks comprise a local exploration of the underlying graph: after treating a node u, the next nodes to be referenced belong to the neighborhood of u. We argue in this thesis that a suitable exploitation of this behavior (through a proper cache management for example) may improve performance. This chapter introduces all the bases needed to understand the contributions presented in chapters 3, 4, 6.

Chapter organization. The chapter starts with complex networks representation at section 2.1. Then, it presents some examples of graph analysis applications at section 2.2. Section 2.3 is dedicated to multi-core architectures with a description of cache management at section 2.3.2. Perf tool is described at section 2.4 with an emphasis in cache-references event at section 2.4.1 and cache-misses event at section 2.4.2. Finally, section 2.5 makes a summary of the entire chapter.

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 representation. This section details each of these representations. Figure 2.1 is an illustration example. It is an undirected weighted graph with 8 nodes.

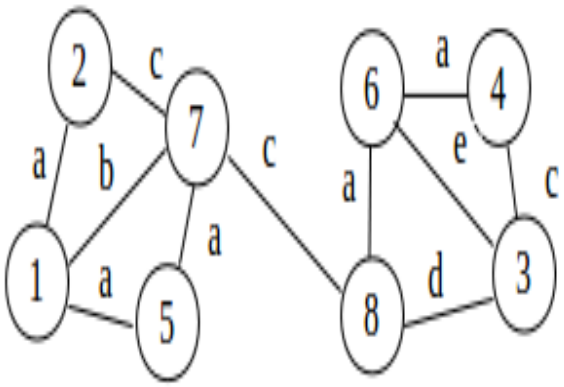


Figure 2.1: Example of graph (G1)

2.1.1 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 2.1, the total used space is 8x8 = 64. But 42 (i.e 66%) are wasted to save zeros.

2.1.2 Yale Representation

The representation often adopted by some graph specific languages such as Galois [33] and Green-Marl [19], is that of Yale [14]. 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 2.1.

2.1.3 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 either to a block of its neighbors [38] (see figure 2.2-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 [13, 4] (see figure 2.2-b)

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 2.1: Yale representation of graph (G1)

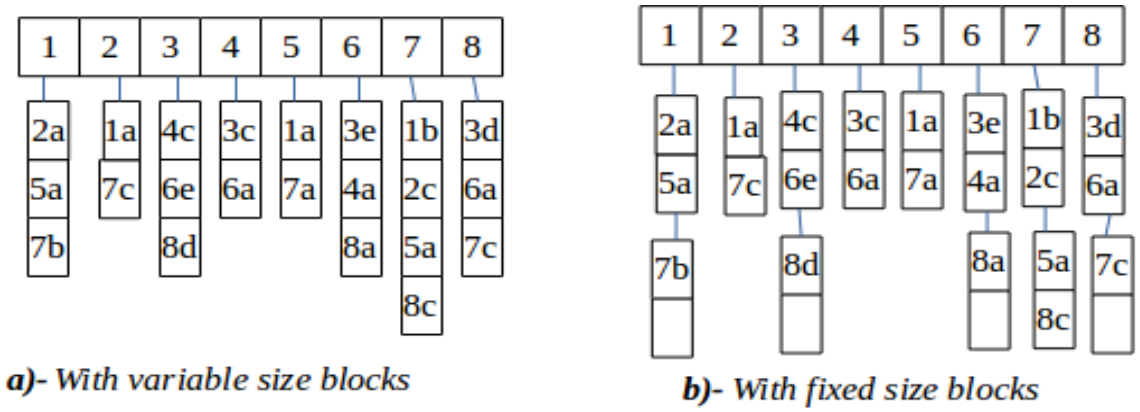


Figure 2.2: Adjacency list representations of (G1)

2.2 Graph Analysis Examples

In this section, we present three graph analysis examples used in this thesis: Pagerank, Katz score and Preferential Attachment score.

2.2.1 Preferential Attachment Score

Preferential Attachment score is also used in link prediction [27]. With this score, the probability that a new edge involves node x is proportional to $|\Gamma(x)|$, the current number of neighbors of x. Preferential Attachment score between two nodes x and y can be computed by the following formula:

$$attach\_score(x, y) = |\Gamma(x)|.|\Gamma(y)| \tag{2.1}$$

Where:

- $\Gamma(x)$ set of x neighbors,
- and $|\Gamma(x)|$ cardinal this set.

For graph (G1) in figure 2.1, Preferential Attachment scores between 1 and each nodes

of the set {3,4,6,8} are follows:

$$\begin{aligned} attach\_score(1,3) &= |\Gamma(1)|.|\Gamma(3)| = 3*3 = 9 \\ attach\_score(1,4) &= |\Gamma(1)|.|\Gamma(4)| = 3*2 = 6 \\ attach\_score(1,6) &= |\Gamma(1)|.|\Gamma(6)| = 3*3 = 9 \\ attach\_score(1,8) &= |\Gamma(1)|.|\Gamma(8)| = 3*3 = 9 \end{aligned}$$

Algorithm 1 describes the Preferential Attachment score between nodes of a whole graph. It is used in this thesis as an illustration in chapter 6.

Algorithm 1 : **Preferential_attachment**($Graph\ G < N, E >$)

```
1: for all x ∈ N do
2:   for all y ∈ N and y ≠ x do
3:     pref_attach ← |G.Neighbor(x)| * |G.Neighbor(y)|
4:     print(x, y, pref_attach)
5:   end for
6: end for
```

2.2.2 PageRank

Pagerank [35] is an algorithm used by Google to classify pages on the web. A Pagerank of a page x is given by the following formula:

$$PR(x) = (1-d) + d \sum_{y \in \Gamma_{in}(x)} \frac{PR(y)}{|\Gamma_{out}(y)|} \tag{2.2}$$

Where:

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

Pagerank is used in chapter 3 and chapter 4 through a posix thread implementation proposed by *Nikos Katirtzis*[1], and chapter 6 through an implementation provided by Galois and Green-Marl authors.

These implementations use either *Adjacency List* or Yale (Galois and Green-Marl) to represent graph.

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

2.2.3 Katz Score

Katz Score [24] can be used (in link prediction [27]) as a similarity measure based on the distances between nodes. Katz score between two nodes x and y is given by following formula:

$$katz\_score(x,y) = \sum_{l=1}^{L} (\beta^l . |paths_{x,y}^{<l>}|) \tag{2.3}$$

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.
- $0 < \beta < 1$. 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):

$$\begin{cases} i = 1 & N_i = \Gamma(x) \\ & L_i[y] = 1, \forall y \in N_i \\ & \\ 2 \leq i \leq L & N_i = \{z / z \in \Gamma(y) \wedge y \in N_{i-1}\} \\ & L_i[z] = \sum_y L_{i-1}[y]_{/\{y \in N_{i-1} \wedge z \in \Gamma(y)\}} \end{cases} \tag{2.4}$$

Katz score is used as an illustration in chapter 5.

The next section presents machine architectures used in this thesis.

2.3 Multi-core Architectures

In this section we first present architectures that will be used in experimental analysis. Then we present cache memory management.

2.3.1 Example of Multi-core Architectures

In this thesis, we used 3 architectures to make our experimental analysis: a machine with 2 cores (Core2), a numa machine with 4 Numa nodes (Numa4) and another machine with 24 Numa nodes (Numa24).

NUMA machines. NUMA stands for Non-Uniform Memory Access. In NUMA machines, time to access to memory depends on the memory location. We use two NUMA nodes machines. The first one with 4 NUMA nodes is referenced as Numa4. The second one with 24 NUMA nodes is referenced as Numa24.

Figure 2.3 shows one of the four nodes of Numa4. Each Numa4 node has 8 CPU cores running at 2.27 GHz. Each core has 32 KB instruction L1, 32 KB data L1 caches and 256KB L2 cache. The 8 cores are sharing 24 MB L3 cache memory and a local memory of size 16 GB. The all architecture has 32 cores and a memory Ram with size 64 GB.

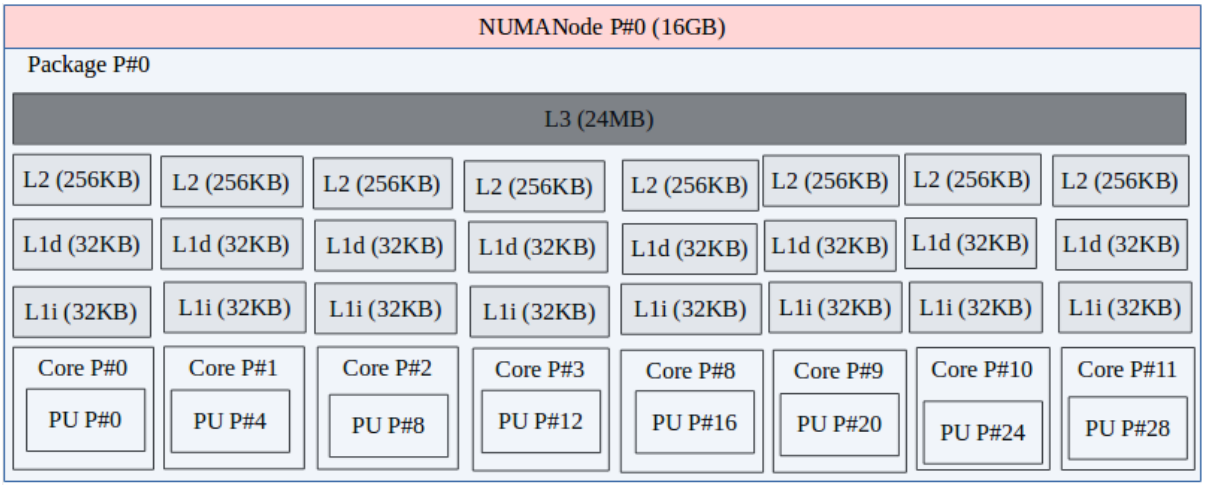


Figure 2.3: a Numa node: 8 cores, private L1 and L2, shared L3, 2.27GHz

A node in Numa24 is similar to the one in Numa4. Each Numa24 node also has 8 CPU cores but running at 2.40 GHz. We have the same size of L1 and L2 caches. Another difference is L3 with size 20 MB. The total number of cores is 192 (that corresponds to 384 threads when Hyper-Threading is enabled). The total memory size is 756 GB.

Core2. Figure 2.4 presents the simplest architecture used in this thesis. It has two cores running at 1.8 GHz. L3 cache memory with 2 MB is shared by the two cores while L1 and L2 caches are private.

All these architectures have hierarchical memory: main memory, cache L3, cache L2 and cache L1. Cache memories greatly influenced architectures performance.

2.3.2 Cache Management

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

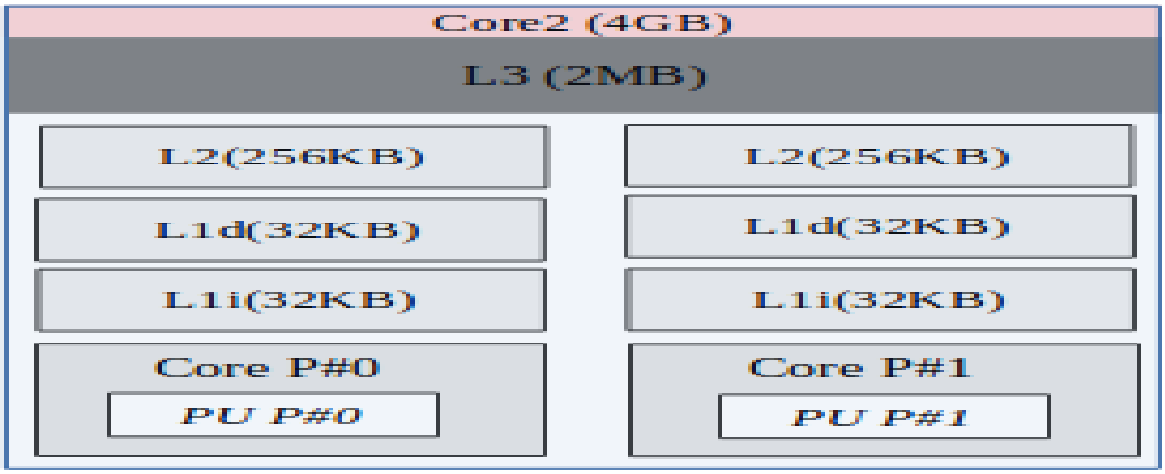


Figure 2.4: Core2 architecture: private L1 and L2, shared L3, 1.80GHz

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. Chapter 3 presents a simple cache model (at section 3.1.3) and shows how complex graphs can be organized in order to reduce cache misses.

In this thesis, measures on architectures (presented at section 2.3.1) are collected with a famous tool, Linux Perf. The next section makes a reminder about this tool.

2.4 Linux Perf Tool

Perf is a profiler tool for Linux systems. It is based on perf-events exported by Linux Kernel. Perf tool offers a rich set of command to collect and analyze performance. The

command line usage is a generic tool that implements a set of commands: stat, record, annotate, etc. In this thesis, we used perf stat for our experimental analysis.

For any supported events (available with *perf list* command), perf can keep a running count during process execution. Perf stat is used to aggregate and present occurrences of selected events. Our experimental analysis are focused on two hardware events: cache-misses and cache-references.

2.4.1 Cache-references Event

This event counts the number of time the last level cache memory is referenced regardless there is a cache miss or not. Consider a memory hierarchy with four level: main memory, cache L3 (last level cache), cache L2 and cache L1. When comparing the execution of two versions of the same application, the one with lesser number of cache references is the one with lesser number of cache misses occurred in L1 and L2 cache memories.

2.4.2 Cache-misses Event

This event counts the number of time the last last level of cache is referenced and there is a cache miss. Consider the memory hierarchy described in the previous paragraph. When comparing the execution of two versions of the same application, the one with a lesser number of cache misses is the one that has a better storage of data in memory.

2.5 Chapter Summary

We saw in this chapter everything that is necessary to understand the next chapters. The elements we presented will be used in those chapters: complex network representation, graph analysis examples (Pagerank, Katz, Preferential Attachment), multi-core architectures (Core2, Numa4, Numa24), cache-management, cache-references and cache-misses events (from Perf tool).

Some of the presented elements will be met in the next chapter. In that chapter, we will introduces the first aspect of our contribution: graph ordering heuristics to ensure cache misses reduction.

Chapter 3

Graph Ordering Heuristics for Cache Misses Reduction

We saw at chapter 1 that complex networks exhibit many properties, among them we have community structure and heterogeneity of node degrees [32]. Due to the huge amount of data, complex network algorithms (graph analysis algorithms) require high-performance computers to extract knowledge and understand the behavior of entities and their relationships in reasonable time. This chapter addresses one of the major challenges of graph analysis [29]: **improving performance**.

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 [2] that a lot of time is wasted in CPU cache latency (with database query processing); in [6] and in [39], the processor is stalled half of the time due to cache misses. Thus, it is often a challenge to write a program that uses efficiently cache memories. Meeting this challenge increases program performances. This challenge is much visible in graph analysis programs due to poor locality, which results from the fact that relationships represented by graphs are often irregular and unstructured [29].

This chapter shows how to use one of the complex-network properties, community structure, in order to improve graph analysis performance by a proper memory management. We present Cn-order, a heuristic that combines advantages of the most recent algorithms (Gorder, Rabbit and NumBaCo) to reduce cache misses in graph algorithms.

Chapter organization. The remainder of this chapter is structured as follows:

- Section 3.1 presents the general problem studied in this chapter.
- Section 3.2, section 3.3 and section 3.4 show the most recent graph ordering heuris-

tics that solve this problem: NumBaCo that we designed and two others Gorder and Rabbit that was designed at the same period to NumBaCo.

- Section 3.5 presents a new graph ordering that combine the advantages of the most recent, to ensure efficient cache management during network analysis.
- Section 3.6 precises how to benefit to our heuristic according to the target graph algorithm complexity.
- Experimental results are presented at section 3.7;
- And section 3.8 is devoted to the synthesis of this chapter.

3.1 Graph Ordering Problem

In this section, in order to present the Graph Ordering Problem, we first show the common pattern in graph analysis and an example. After that we present the cache model, then the problem itself and its complexity. And finally we show how we will measure graph ordering efficiency that distinguishes two different graph orderings.

3.1.1 Common Pattern in Graph Analysis

One common statement used in graph analysis is as follows:

```
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 that successive u and v are close in memory.

3.1.2 Illustration Example

Figure 3.1 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

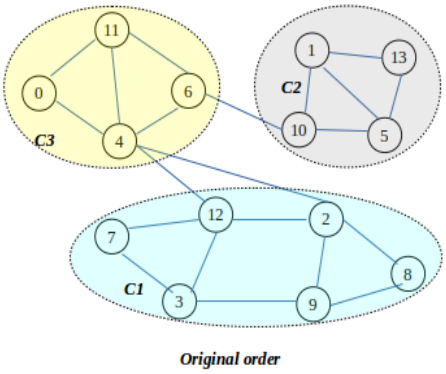


Figure 3.1: Running example

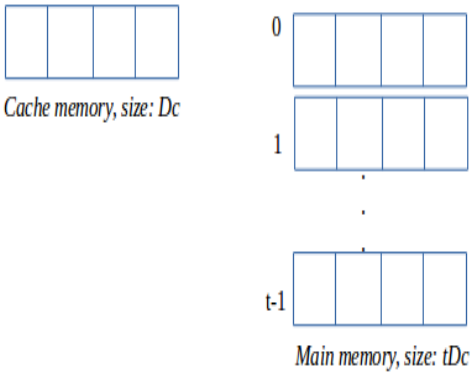


Figure 3.2: Memory representation

the data cache provide in figure 3.2, 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 in figure 3.4, 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.

3.1.3 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 3.2).

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.

A cache miss can be modeled by σ as follows:

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

3.1.4 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:

$$CacheMiss(N_k) = 1 + \sum_{i=2}^{k} \sigma(n_i, n_{i-1}) \tag{3.1}$$

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_{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 3.1.1 (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 [17]. As reminder, this problem is defined as follows:*

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

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.

3.1.5 Graph Ordering Efficiency

As presented in the previous paragraphs, NGP and OLAP are closed problems. In order to measure graph ordering efficiency, in addition to cache miss ($CacheMiss()$ described at equation 3.1), we will use nodes closeness defined as:

$$nodes\_closeness = \sum_{\{n_i, n_j\} \in E} |\pi(n_i) - \pi(n_j)| \tag{3.2}$$

One should notice that *nodes_closeness* depends only on the graph. But $CacheMiss(N_k)$ depends on both graph and the program that runs this graph. At performance analysis in section 3.7, *nodes_closeness* is used as it is, but for $CacheMiss(N_k)$, Linux Perf tool is used.

The next section described existing heuristics to solve NGP.

3.2 Early Graph Ordering Heuristics

In this section, we present two recent algorithms: Gorder [53] and Rabbit Order [3].

3.2.1 Gorder

This paragraph presents Gorder idea, key function, algorithm, advantages and disadvantages.

Gorder idea

The goal is to reduce cache misses by making sibling nodes to be close in memory (allow them to fit in cache memory). Consider the graph in figure 3.1. $Neig(1) = \{5, 10, 13\}, Neig(5) = \{1, 10, 13\}$. Let us consider the data cache provided in figure 3.2.

From figure 3.1, 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 3.4, 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 order to minimize the cpu cache miss ratio.

Hao Wei *et al.* [53] 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 and 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 a unique number $[1, n]$, $n = |N|$ is the number of nodes.

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

Gorder algorithm

Algorithm 2 presents Gorder. Hao Wei *et al.* [53] 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 a way that $\sum_{j=max\{1,i-w\}}^{i-1} S(v_i, v_j)$ is maximal.

Algorithm 2 : **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 us consider the problem of finding frequent itemsets, itemsets are sets of neighbors; choosing the best v_i can be seen as choosing the frequent itemsets 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 the performances will be if we consider frequent itemsets with size more than 2.

3.2.2 Rabbit Order

The goal of Rabbit is to make nodes that belong to the same sub-community to be close in the memory. Algorithm 3 presents Rabbit order.

- Line 1 uses parallel version of GCA (described at section 3.3.1) to produce communities. This parallel version was introduced in [3] in order to ensure just in time graph ordering.
- Line 2 of Rabbit order visits each community in DFS fashion and assigns a new number to each node. This action makes nodes belonging to the same sub-community to be close in the memory.

Algorithm 3 : **Rabbit Order**

Input: $G = (V, E)$

Output: π a permutation on V nodes

1: $Com \leftarrow parallel\_GCA\_communities\_detection(G)$
2: $\pi \leftarrow graph\_numbering(Com, G)$
3: **return** π

Figure 3.4 gives rabbit order of graph in figure 3.1. Nodes that belong to the same sub-community are consecutive, for example $\pi\{0, 4, 6, 11\} = \{0, 1, 2, 3\}$.

3.3 Numbaco

The main idea of this order is to make nodes that belong to the same community to be consecutive. In that way, nodes belonging to the same community will be close in memory.

3.3.1 Community detection algorithm

Both NumBaCo and Rabbit orders start by detecting communities in a graph before numbering. If NumBaCo uses Louvain algorithm [7], Rabbit order uses GCA (Graph Clustering Algorithm) [43] that can be seen as a light version of Louvain.

Louvain algorithm. Louvain algorithm [7] 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.

Algorithm 4 details community detection with Louvain. It starts with a partition where each node is alone in its community (Lines 2 to 4). Then, it computes communities by repeating iteratively the two following phases:

> ***Step 1)*** For each node u, evaluate the gain in modularity that may be obtained by removing u from its current community and placing it in the community of a neighbor v. Place u in the community for which this gain (if positive) is maximum. Each node may be considered several times. (Lines 5 to 14).
>
> ***Step 2)*** Generate a new graph where nodes are communities detected in the first phase. Weights of links between new nodes are given by summing weights of links between nodes in the corresponding communities. Reapply ***Step 1*** to the resulting weighted network.

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

GCA. In contrast to Louvain algorithm, GCA (detailed in algorithm 5) does not traverse all vertices multiple times and thus contributes to reduce drastically execution time. For this purpose, it introduces three techniques:

- It incrementally prunes vertices whose clusters are obtained without modularity computing (lines 4 to 8 in Algorithm 5).

Algorithm 4 : *Louvain_communities_detection* [7]

Input: $G = (V, E), max\_mod$: a non-oriented graph and the maximum modularity

Output: Hierarchical communities of G

1: **repeat**
2: **for** $u \in V$ **do**
3: $C_u \leftarrow u$ *//each node is alone in its community*
4: **end for**
5: **repeat**
6: $migrated \leftarrow false$
7: **for** $u \in V$ **do**
8: $v \leftarrow arg \max_{v' \in neig(u)} \Delta Q_{C_u C_{v'}}$ *//finding the best community to move u*
9: **if** $(\Delta Q_{C_u C_v} > 0)$ **then**
10: move u from C_u to C_v
11: $migrated \leftarrow true$
12: **end if**
13: **end for**
14: **until** $(migrated = false)$
15: $G \leftarrow build\_graph(G, \{C_1, C_2, ..., C_m\})$ *//Using detected communities to build a new graph*
16: $mod \leftarrow modularity(G)$
17: **until** $(mod \geq max\_mod)$

- It reduces the number of modularity computations by selecting at each step of clustering process a node u that has the smallest degree (line 9 in Algorithm 5).

- It incrementally aggregates vertices which are placed in a same cluster into a single vertex to eliminate unnecessary vertices from the graph (lines 11 to 17 in Algorithm 5).

Louvain algorithm and GCA generate 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 whith graph in figure 3.4, is given at figure 3.3. 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.1 with karate graph.

Algorithm 5 : *GCA_communities_detection* [43]

Input: $G = (V, E)$: a non-oriented graph

Output: Hierarchical communities of G

1: $i \leftarrow 0, P_0 \leftarrow \emptyset, T \leftarrow V$;
2: **while** $|T_i| > 0$ **do**
3: $i \leftarrow i + 1$;
4: $P_i \leftarrow \{u : |\Gamma(u)| = 1\}$; *// P_i contains nodes using for pruning*
5: **for** $u \in P_i$ **do**
6: aggregate u into its neighbor;
7: **end for**
8: $T_i \leftarrow T_{i-1} - P_i$;
9: Select $u \in T_i$ with the smallest degree;
10: $v \leftarrow arg \max_{v' \in neig(u)} \Delta Q_{uv'}$ *//finding the best destination for u*
11: **if** $(\Delta Q_{uv} > 0)$ **then**
12: aggregate u and v into a single w; *//Incrementally aggregation*
13: $T_i \leftarrow T_i - \{u, v\}$;
14: $T_i \leftarrow T_i \cup \{w\}$;
15: **else**
16: $T_i \leftarrow T_i - \{u\}$;
17: **end if**
18: **end while**

3.3.2 NumBaCo Algorithm

Algorithm 6 presents NumBaCo order.

- The first line of NumBaCo generates communities with Louvain algorithm (see Algorithm 4).
- The second line classifies communities in order to bring together ones with higher affinity (number of edge shared).
- Line three assigns numbers in such a way that nodes belonging to the same community are consecutive. Community j follows community i if community j has the highest affinity for all j > i.
- Line 4 changes neighbors position according to their ascending new orders: first store neighbors that have the smaller orders and then the others with bigger orders.

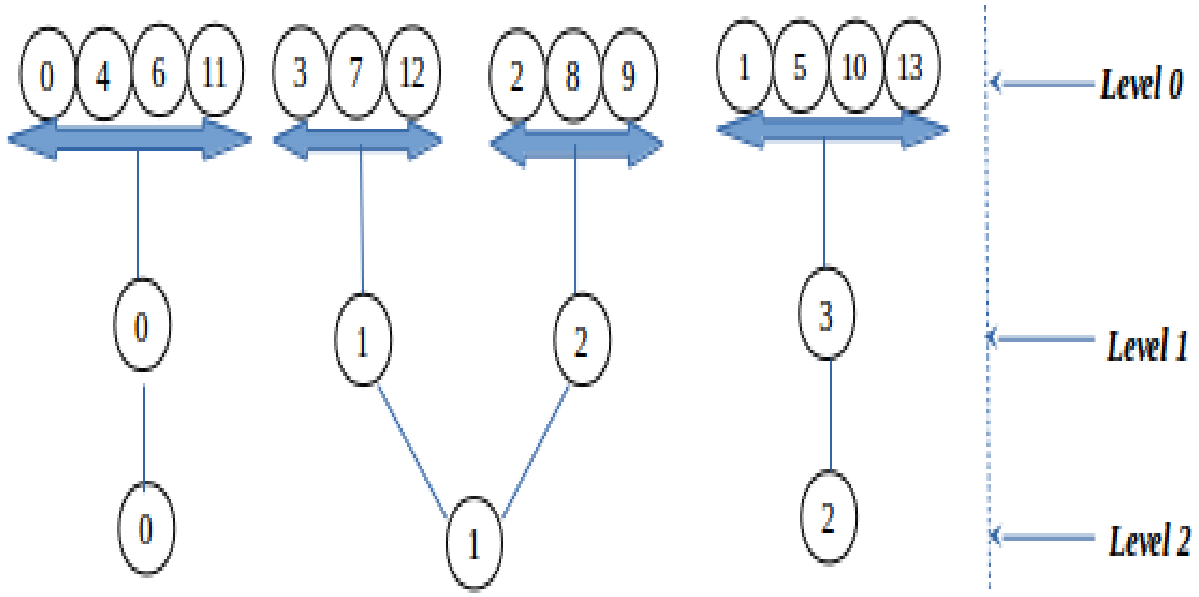


Figure 3.3: Running graph with Louvain

Algorithm 6 : **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: $G' \leftarrow store\_neighboors(G, \pi)$
5: **return** G'

3.3.3 Differences between NumBaCo and Rabbit

As already mentioned before, NumBaCo and Rabbit orders are based on communities detection. However, there are some differences between them.

- After detecting communities, Rabbit generates directly a numbering. But NumBaCo first classifies communities to make ones with higher affinity (number of edge shared) being together, then it performs the numbering and changes neighbors storage.

- 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 belonging to the same sub-community to 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 a way that $\pi(x) < \pi(y)$.

For graph example in figure 3.1, $\pi\{2, 3, 7, 8, 9, 12\}$ gives $\{4, 5, 6, 7, 8, 9\}$ in figure 3.4 with NumBaCo and $\{7, 4, 5, 8, 9, 6\}$ with Rabbit.

3.4 Advantages and Disadvantages of Heuristics

This section presents advantages and disadvantages of each presented heuristic: Gorder, Rabbit and NumbaCo. These three heuristics were published at the same period; each of them claims to outperform state of the art algorithms such as: BFS order [21], Graph partition order with METIS [23], SlashBurn [28, 20], RCM [21, 11], Nested Dissection [26], LLP [8], Shingle [10], Minla [36, 40], Mloga [41], Chdfs [5], LDG [49]. This section presents strengths and weaknesses of each heuristic.

3.4.1 Advantage and Disadvantage of Gorder

The main advantage of gorder is that it brings closer pairs of nodes appearing frequently in direct neighborhood. However, 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 A.2 with karate graph.

3.4.2 Advantage and Disadvantage of Rabbit.

The main advantage of Rabbit is that it allows nodes belonging to the same sub-community to be closer in memory. Nothing is done to place sub-communities or communities. One way to improve this order is by classifying communities or sub-communities by affinities: this is because some communities or sub-communities share many links compared to others.

3.4.3 Advantage and Disadvantage of NumbaCo

The main advantage of NumBaCo is that it allows nodes belonging to the same community to be closer in memory. It also ensures that communities with higher affinities are

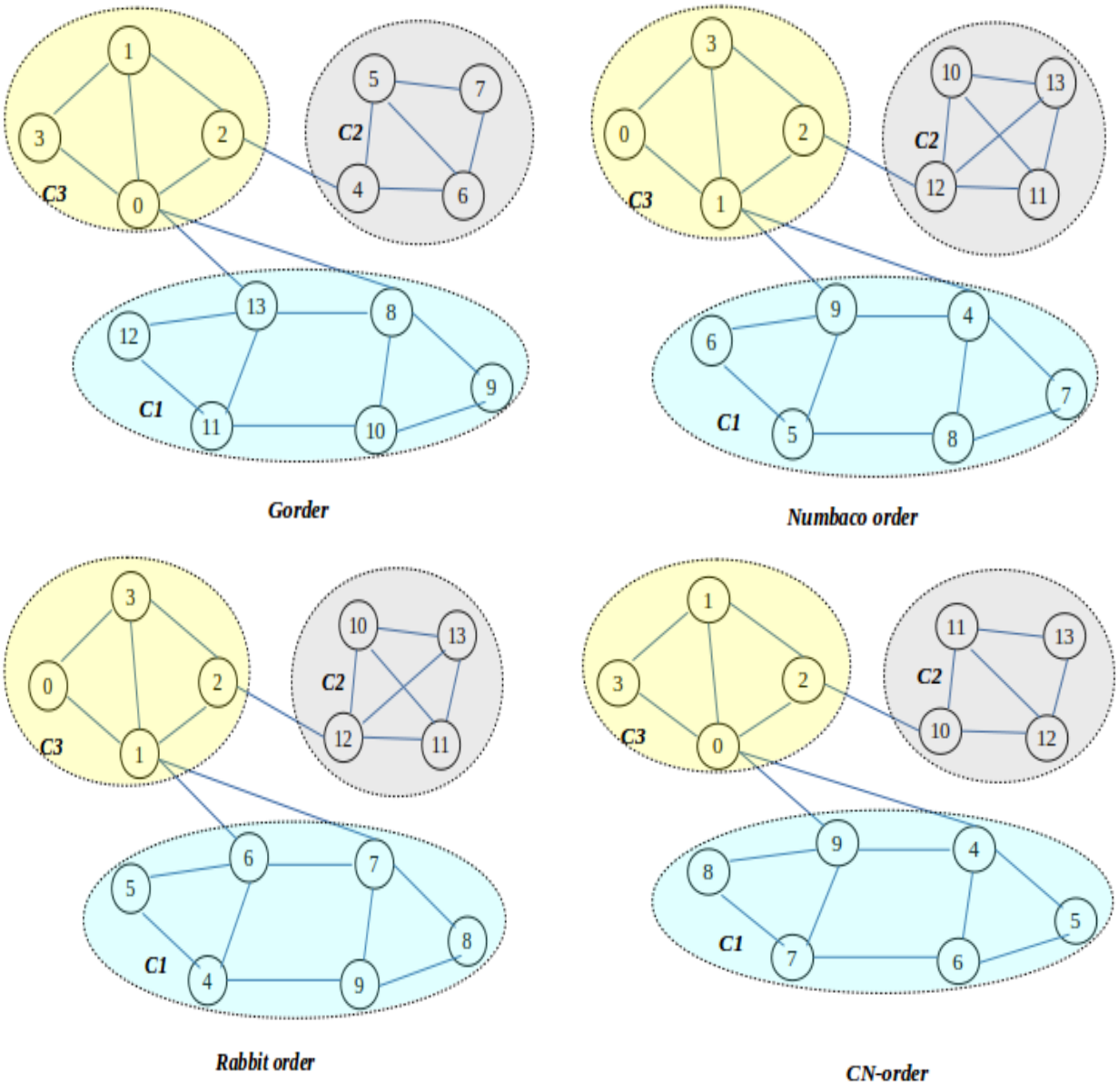


Figure 3.4: Example of graph with different orders

close in the memory. The disadvantage is that there is no particular numbering within a community (within community, even if nodes are consecutive, they follow the initial order they had before using NumBaCo). A good numbering within communities could allow better performances.

3.4.4 Strengths and Weaknesses Summary

Tables 3.1 and 3.2 group strengths and weaknesses of the three studied order heuristics. These are essentially advantages and disadvantages already presented previously.

Table 3.1: Strengths

Heuristics	**Strengths**
Gorder	– Brings closer sibling nodes
Rabbit	– Brings closer nodes of same sub-communities
NumBaCo	– Brings closer nodes in same community – Brings closer communities with higher affinities

Table 3.2: Weaknesses

Heuristics	**Weaknesses**
Gorder	– Scatters communities or sub-communities – Is not architecture-aware
Rabbit	– No strategy to place (sub-)communities according to their affinities – Is not architecture-aware
NumBaCo	– No strategy to assign numbers to nodes withing a community – Is not architecture-aware

Note that, all the three have one common weakness, they are not architecture-aware. The next paragraph shows how these strengths and weaknesses are used to build a more powerful graph order called Cn-order.

3.5 Cn-order: a new Complex Network order

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 7.

3.5.1 Cn-order Version 1

Algorithm 7 : **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: $G' \leftarrow store\_neighboors(G, \pi)$

— Line 1 computes π_1 in Gorder fashion. Details are close to algorithm 10 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 taken into account): — line 2 computes communities with Louvain algorithm [7]; — Line 3 classifies communities according to their affinities (algorithm 8, line 1 to 6); it also delimits communities in different groups in such a way that each group will fit into cache memory with the size $Cache\_size$ (algorithm 8, line 7 to 20).

— 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

Algorithm 8 : $classify\_and\_find\_offsets$

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: $C_{max} \leftarrow arg \max_{\forall Com_r[k]} |\{(u,v) : u \in Com_{cl}[i-1], v \in com_r[k]\}|$
5: $Com_{cl}[i] \leftarrow C_{max}$; $Com_r \leftarrow Com_r - \{C\}$; $i \leftarrow i+1$
6: **end while**
7: $i \leftarrow 1$; $offset \leftarrow 0$
8: **for all** $C \in Com_{cl}$ **do**
9: **if** $C.nb\_edge \leq Cache\_size$ **then**
10: $Com_{class}[i] \leftarrow C$,
11: $Com_{class}[i].offset \leftarrow offset$,
12: $i \leftarrow i+1$; $offset \leftarrow offset + C.nb\_node$
13: **else**
14: **for all** *subcommunity* $s\_C \in Com_{cl}$ **do**
15: $Com_{class}[i] \leftarrow s\_C$,
16: $Com_{class}[i].offset \leftarrow offset$,
17: $i \leftarrow i+1$; $offset \leftarrow offset + s\_C.nb\_node$
18: **end for**
19: **end if**
20: **end for**

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 9).

3.5.2 Cn-order Version 2

The goal here is to reduce execution time and keep 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 [43] and precisely a parallel communities detection based on this algorithm proposed in [3].

Algorithm 9 :**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** $i \in [1 \ldots nb_{class}]$ **parallel do**
5: $\quad compute\_community\_order(Com_{cl}[i], G, \pi)$
6: **end for**
7: $G' \leftarrow store\_neighboors(G, \pi)$

Line 5 of cn-order (detailed in Algorithm 10) generated order for each community. We adapted Gorder [53] to be applied in each community (each community is seen as a small graph).

Algorithm 10 : *compute_community_order* (Gorder [53] 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**

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).

3.6 Time Complexity Analysis

All the heuristics Gorder, Rabbit, NumBaCo, Cn-order are intended to be used before running graph algorithms. This is to ensure a better storage of graph by increasing memory locality. According to the target graph algorithm time complexity, these heuristics can be used either for graph preprocessing or during graph loading in program execution. This section discusses heuristics time complexity in order to find out how to benefit from them.

3.6.1 Graph Algorithms Categories

Let GA be the set of all graph algorithms. When considering both heuristic H and graph algorithm A, the total time complexity is: $cplx(H) + cplx(A)$. We distinguish two categories of graph algorithms:

- $Alg_1(H) = \{A \in GA \mid cplx(A) > \ cplx(H)\}$: graph algorithms that have higher complexity than H.
- $Alg_2(H) = \{A \in GA \mid cplx(A) \leq \ cplx(H)\}$: graph algorithms that have lesser complexity than H.

3.6.2 Benefit From Heuristics

Graph algorithms that belong to $Alg_1(H)$ will obviously benefit from performance improvement due to heuristic H, because the time to re-order the graph is negligible compared to the time taken to process these graph algorithms. In this case, H can be used either for graph preprocessing or during graph loading in program execution. But for graph algorithms that belong to $Alg_2(H)$, the only way to benefit from H is, using it for graph preprocessing.

3.6.3 Time Complexities

Consider a graph $G = (N, E)$, where $n = |N|$ and $m = |E|$. In Table 4.1, $cplx(H)$ is the time complexity of heuristic H (Gorder, Rabbit, NumBaCo or Cn-order). Among the first three, Gorder is the slowest. Rabbit or NumBaCo time complexity are due to the communities detection part. In practice, rabbit time is reduced because communities are detected in parallel. Cn-order time complexity is the sum of Gorder and NumBaCo time complexities.

In order to reduce Cn-order time complexity, one can increase parallelization or can tune communities detection part. This can be done by dividing graph nodes into small groups (communities for example) and applying Gorder to each small group in parallel. It is for example what we did in the second version of Cn-order.

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

Table 3.3: Time complexity comparison

Table 4.1 shows that:

$$cplx(\text{Rabbit}) < cplx(\text{NumBaCo}) < cplx(\text{Gorder}) < cplx(\text{Cn-order})$$

If we consider Rabbit and Gorder for example, this observation implies that:

- $|Alg_1(Rabbit)| > |Alg_1(Gorder)|$: the number of graph algorithms that belong to $Alg_1(Rabbit)$ is greater than the number of graph algorithms that belong to $Alg_1(Gorder)$. In other words, there are more graph algorithms with higher time complexity compared to Rabbit than graph algorithms with higher time complexity compared to Gorder.
- $|Alg_2(Rabbit)| < |Alg_2(Gorder)|$: the number of graph algorithms that belong to $Alg_2(Rabbit)$ is smaller than the number of graph algorithms that belong to $Alg_2(Gorder)$. That is, there are more graph algorithms with lesser time complexity compared to Gorder than graph algorithms with lesser time complexity compared to Rabbit.

In this section, we showed that regardless of heuristic time complexity compared to target graph algorithm, we can always benefit from them, either using them for preprocessing or during graph loading in program execution. The following section presents the performances gained from these heuristics.

3.7 Experimental Analysis

For this evaluation, we present results got with the well known graph analysis application, Pagerank [35]. We used a posix thread implementation proposed by *Nikos Katirtzis*[1]. This implementation uses adjacency list representation. The experiments were made with one thread on architectures described in Chapter 2 (Core2, Numa4 and Numa24).

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

In these experiments, we used four performance elements: Nodes Closeness, execution time, cache references and cache misses. The last two were reported by Linux Perf tool (also described in Chapter 2).

3.7.1 Graph Ordering Comparison

Results in tables 3.4, 3.5,3.8, 3.7, 3.6 , 3.9 confirm the strengths and weaknesses of each algorithm presented in section 3.4. These results also confirm that Cn-order uses the advantages of all the first three graph ordering:

- 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 outperforms all the orders, since it combines all the advantages.
- The best $Nodes\_closeness$ is switching between Cn-order and Numbaco. This is because Cn-order results from Numbaco (that gives the best) and Gorder (that gives the worst, with a negative gain).

Comparison with Live Journal dataset

This dataset [54] is unoriented graph with 3,997,962 nodes and 34,681,189(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 of any studied architecture).

In tables 3.4, 3.5,3.6:

- Compared to Rabbit and Gorder, Numbaco has the best performances in terms of $Nodes\_closeness$, cache misses and time reduction.
- Gorder allows to have least references to L3 than Rabbit and NumBaCo.
- Except for the $Nodes\_closeness$, CN-order outperforms all the orders:

- With Core2,
  - it reduced time by 17.52% compared to original, 8.15% compared to Gorder, 7.9% compared to Rabbit and 3.36% compared to Numbaco.
  - It reduced cache misses by 36.24% compared to original, 16.17% compared to Gorder, 11.65% compared to Rabbit and 5.17% compared to Numbaco.
  - It reduced cache references by 29.68% compared to original, 5.2% compared to Gorder, 14.27% compared to Rabbit and 19.19% compared to Numbaco.
- With Numa4,
  - it reduced time by 29.60% compared to original, 19.63% compared to Gorder, 8.91% compared to Rabbit and 2.04% compared to Numbaco.
  - It reduced cache misses by 46.09% compared to original, 35.4% compared to Gorder, 10.25% compared to Rabbit and 1.07% compared to Numbaco.
  - It reduced cache references by 33.38% compared to original, 7.33% compared to Gorder, 14.03% compared to Rabbit and 9.96% compared to Numbaco.
- With Numa24,
  - it reduced time by 22.06% compared to original, 14.11% compared to Gorder, 7.08% compared to Rabbit and 1.84% compared to Numbaco.
  - It reduced cache misses by 45.30% compared to original, 34.79% compared to Gorder, 9.12% compared to Rabbit and 00.73% compared to Numbaco.
  - It reduced cache references by 32.87% compared to original, 7.43% compared to Gorder, 16.49% compared to Rabbit and 9.83% compared to Numbaco.

Comparison with Orkut dataset

This dataset [54] is unoriented graph with 3,072,441 nodes and 117,185,083(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*117,185,083 + 3,072,441)*4bytes = 905.77 MB (do not fit in L3 cache).

We have quite the same observations in tables 3.7, 3.8,3.9:

- Except for the *Nodes_closeness*, CN-order outperforms all the orders Gorder, Rabbit Numbaco:

Table 3.4: Performances comparison (Pagerank, Live Journal, 1 thread, core2)

Heuristic	Original	Gorder	Rabbit	NumBaCo	**cn-order**
Time (s)	241.975	219.280 (09.37%)	218.685 (09.62%)	207.707 (**14.16%**)	199.568 (**17.52%**)
L3 cache miss	5,490	4,388 (20.07%)	4,140 (24.59%)	3,784 (**31.07%**)	3,500 (**36.24%**)
L3 cache ref	6,572	4,963 (**24.48%**)	5,559 (15.41%)	5,291 (19.49%)	4,621 (**29.68%**)
nodes_closeness	38,631	41,407 (+**07.18%**)	26,390 (31.68%)	17,922 (**53.60%**)	19,734 (48.91%)

Table 3.5: Performances comparison (Pagerank, Live Journal, 1 thread, Numa4)

Heuristic	Original	Gorder	Rabbit	NumBaCo	**cn-order**
Time (s)	345.297	310.851 (09.97%)	273.832 (20.69%)	250.313 (**27.56%**)	243.070 (**29.60%**)
L3 cache miss	3,356	2,997 (10.69%)	2,153 (35.84%)	1,845 (**45.02%**)	1,809 (**46.09%**)
L3 cache ref	7,689	5,686 (**26.05%**)	6,201 (19.35%)	5,888 (23.42%)	5,122 (**33.38%**)
nodes_closeness	38,631	41,407 (+**07.18%**)	26,390 (31.68%)	17,922 (**53.60%**)	18,349 (52.50%)

- With Core2,
  - it reduced time by 19.38% compared to original, 4.05% compared to Gorder, 8.89% compared to Rabbit and 7.4% compared to Numbaco.
  - It reduced cache misses by 33.36% compared to original, 5.2% compared to Gorder, 15.56% compared to Rabbit and 14.31% compared to Numbaco.
  - It reduced cache references by 25.25% compared to original, 00.24% compared to Gorder, 21.96% compared to Rabbit and 17.93% compared to Numbaco.
- With Numa4,
  - it reduced time by 36.89% compared to original, 17.13% compared to Gorder, 7.42% compared to Rabbit and 6.12% compared to Numbaco.
  - It reduced cache misses by 47.94% compared to original, 29.54% compared

Table 3.6: Performances comparison (Pagerank, Live Journal, 1 thread, Numa24)

Heuristic	Original	Gorder	Rabbit	NumBaCo	**cn-order**
Time (s)	209.47	192.8 (07.95%)	178.077 (14.98%)	167.107 (**20.22%**)	163.247 (**22.06%**)
L3 cache miss	3,587	3,210 (10.51%)	2,289 (36.18%)	1,988 (**44.57%**)	1,962 (**45.30%**)
L3 cache ref	7,671	5,719 (**25.44%**)	6,414 (16.38%)	5,903 (23.04%)	5,149 (**32.87%**)
nodes_closeness	38,631	41,407 (+**07.18%**)	26,390 (31.68%)	17,922 (**53.60%**)	19,517 (49.47%)

to Gorder, 5.34% compared to Rabbit and 5.02% compared to Numbaco.

* It reduced cache references by 30.20% compared to original, 02.02% compared to Gorder, 20.55% compared to Rabbit and 16.1% compared to Numbaco.

– With Numa24,

* it reduced time by 28.22% compared to original, 12.07% compared to Gorder, 8.87% compared to Rabbit and 6.82% compared to Numbaco.
* It reduced cache misses by 47.65% compared to original, 27.7% compared to Gorder, 10.03% compared to Rabbit and 07.93% compared to Numbaco.
* It reduced cache references by 30.75% compared to original, 02.20% compared to Gorder, 21.85% compared to Rabbit and 16.85% compared to Numbaco.

3.7.2 Effect of Architectures in Performances

The general observation is that performance does not vary exactly in the same direction from one architecture to another: if cn-order produces almost always the best performance (reduced time, reduced cache misses and reduced cache references) in all the three architectures and with the two datasets; performance does not vary in the same way with other orders (Gorder, Rabbit, Numbaco) on each architecture.

In order to understand how performance varies according to heuristics, one should observe the "original" behavior . It is easy to observe that: time, cache misses, cache references, nodes closeness vary from one architecture to another and from one dataset to another.

Table 3.7: Performances comparison (Pagerank, Orkut, 1 thread, core2)

Heuristic	Original	Gorder	Rabbit	NumBaCo	**cn-order**
Time (s)	556.418	471.092 (15.33%)	498.017 (10.49%)	489.715 (**11.98%**)	448.537 (**19.38%**)
L3 cache miss	15,857	11,391 (28.16%)	13,034 (17.80%)	12,836 (**19.05%**)	10,567) (**33.36%**)
L3 cache ref	19,040	14,278 (**25.01%**)	18,413 (03.29%)	17,645 (07.32%)	14,232 (**25.25%**)
nodes_closeness	148,080	151,204 (+**02.10%**)	89,596 (39.49%)	43,735 (**70.46%**)	43,078 (79.90%)

Table 3.8: Performances comparison (Pagerank, Orkut, 1 thread, Numa4)

Heuristic	Original	Gorder	Rabbit	NumBaCo	**cn-order**
Time (s)	831.402	667.068 (19.76%)	586.382 (29.47%)	575.520 (**30.77%**)	524.619 (**36.89%**)
L3 cache miss	8,163	6,661 (18.40%)	4,685 (42.60%)	4,659 (**42.92%**)	4,249 (**47.94%**)
L3 cache ref	22,659	16,272 (**28.18%**)	20,472 (9.65%)	19,463 (14.10%)	15,814 (**30.20%**)
nodes_closeness	148,080	151,204 (+02.10%)	89,596 (39.49%)	43,735 (**70.46%**)	50,448 (65.93%)

Variation of Time

The variation in time is due to the differences between architectures: Core2 (frequency 1.8 GHz, size L3 2 MB) and two Numa machines (2.4 GHz with Numa24 > 2.27 GHz with Numa4, L3 with sizes 20 MB and 24 MB respectively).

Cn-order has the best time reduction in all architectures with the two datasets. Numbaco has the second time reduction in all architectures except in Core2 with Orkut dataset where Gorder comes in the second position. Rabbit is the third in the two Numa machines but it is at the last position in Core2 with both Orkut and Live Journal datasets.

The observation in Core2 is due to three reasons:

- Core2 has a small L3 size compared to the others architectures.
- Gorder behavior: it brings closer pairs of nodes appearing frequently in direct neighborhood. The generated subsets are suitable for Core2 architecture.

Table 3.9: Performances comparison (Pagerank, Orkut, 1 thread, Numa24)

Heuristic	Original	Gorder	Rabbit	NumBaCo	**cn-order**
Time (s)	499.069	418.439 (16.15%)	402.487 (19.35%)	392.253 (**21.40%**)	358.220 (**28.22%**)
L3 cache miss	9,130	7,308 (19.95%)	5,695 (37.62%)	5,503 (**39.72%**)	4,779 (**47.65%**)
L3 cache ref	22,774	16,270 (**28.55%**)	20,744 (8.9%)	19,603 (13.9%)	15,770 (**30.75%**)
nodes_closeness %	148,080	151,204 (+02.10%)	89,596 (39.49%)	43,735 (**70.46%**)	40,460 (**72.67%**)

- The size of dataset: Orkut dataset is bigger than Live Journal dataset. Compared to Rabbit, Numbaco behavior makes it resist with Live Journal but not with Orkut.

Cn-order stays the best even in Core2 because it uses partially Gorder strategy. But time reduction is nothing else than consequence of cache misses reduction, cache references and nodes closeness.

Variation of Cache Misses

In every architecture and every dataset, Cn-order produces the best cache misses reduction. It is follow by Numbaco. Rabbit comes at the third position and Gorder is the last. It is true that in Core2, cache misses reduction produced by Gorder is close to the other orderings compared with what is seen in Numa4 and Numa24. This is because of the three reasons mentioned at section 3.7.2.

Variation of Cache References

After Cn-order, Gorder produces the best cache references reduction for every architecture and every dataset. This is the expected observation according to the way each heuristic works.

Variation of Nodes Closeness

As already mentioned at section 3.1.5, in contrast to the other performance elements that depend on both graph and the program that runs this graph, *Nodes_closeness* depends only on the graph. But for Cn-order, it also depends on the architecture. This is because it generates an order that takes into account the cache memory size of the

target architecture. This is the reason why from one architecture to another, with the same dataset, $Nodes\_closeness$ changes only for Cn-order.

Between the three first orders, Numbaco is the best and Gorder is the last (with even negative gain compared to original). On contrary to the other performance elements, with $Nodes\_closeness$, Cn-order is switching the first position with Numbaco. This is because Cn-order combine two strategies situated at extreme positions: Numbaco strategy and Gorder strategy.

Someone will wander why Cn-order shares its first position in that case (nodes closeness), and not with the other performance elements (cache misses, cache references), since it still combines Gorder and Numbaco strategy. The reason is that, $Nodes\_closeness$ is the only performance that give a negative gain for one of the used strategies (the other performance always produces positive gain for Gorder or Numbaco).

3.8 Discussion

Chapter Assessment. In this chapter, we propose Cn-order, a heuristic that combines the 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).

Experimental results with one thread on Core2, Numa4 and Numa24 (with Pagerank and livejournal for example) showed that Cn-order uses well the advantages of the recent orders and outperforms them.

Chapter Perspectives. This work can be directly improved through:

- Gorder, by using item-set strategy.
- NumBaCo and Rabbit, by using local communities detection to assign numbers inside the community; by comparing existing communities detection algorithms in order to see which one is the best to graph ordering for cache misses reducing. In this chapter, we consider only GCA and Louvain communities detection algorithm.

This chapter focuses on graph ordering for cache misses reduction. In order to design new heuristics or to improve existing one, it will be good to compare theoretically and experimentally graph ordering for cache misses reduction with graph ordering for compression.

Next Chapter. In this chapter, the hole potentiality of each architecture was not used. Our analysis were done with only one thread. In the next chapter, we will study what happens with multiple threads: the influence of orderings when we increase the number of threads and how the workload evolves among threads?

Chapter 4

Degree-aware Scheduling for Load Balancing

The irregular structure of complex networks combined with the spatial locality of graph exploration algorithms, make the scheduling task difficult. 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 with a proper scheduling where threads that are executed together on a parallel computer have balanced load [47, 48].

In this chapter, we introduce a technique that takes into account another complex-network properties, the heterogeneity of nodes degrees to tackle load balancing among threads. This leads to a heuristic named deg-scheduling that, combined with graph ordering heuristics presented in the previous chapter, allows to ensure both cache misses reducing, and load balancing among threads.

Chapter organization. The remainder of this chapter is structured as follows:

- Section 4.1 presents the problem studied in this chapter.
- Section 4.2 shows Deg-scheduling, an heuristic that solves this problem.
- Section 4.3 presents other heuristics that combine Deg-scheduling to graph ordering heuristics to ensure both cache misses reduction and load balancing among threads.
- Section 4.4 precises the complexity of the designed heuristics and it also precises how to benefit to these heuristics according to the target graph algorithm complexity.

- Experimental analysis are presented at section 4.5.
- And section 4.6 is devoted to the synthesis of this chapter and the direct perspectives.

4.1 Scheduling Problem in Graph Applications

Before presenting the scheduling problem that arises in parallel graph applications, we are going to show the parallel version of common pattern presented at Chapter 3.

4.1.1 Common Pattern in Parallel Graph Analysis

There are two cases depending on the number of tasks received by each thread during the execution of a parallel graph application:

- In the first case, each thread receives the same number of tasks. In this case, the scheduling is static. This is usually the case when tasks have the same cost.
- In the second case, the cost is different from one task to another. The number of tasks received by each thread should also be different in order to ensure load balancing among threads. In this case, the scheduling is dynamic.

Common Pattern for Static Scheduling

Each thread th_t (spawned at line 3) will get exactly the same number of tasks. In the following pattern, this number is 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** $t \leftarrow 1$ *to* nb_{thread} **do**
2: $s \leftarrow task\_load[t].start$; $e \leftarrow task\_load[t].end$
3: $spawn\_thread(th_t, \textbf{task\_func}, param(s, e))$
4: **end for**

Common Pattern for Dynamic Scheduling

In this case, each thread may have a number of tasks different to the number of tasks received by the other threads.

1: **global** $setOfTask = \{1, ..., t\}$
2:
3: **do_work**()

4: **while** $setOfTask \neq \emptyset$ **do**
5: $t \leftarrow$ **atomic_take_task**$(setOfTask)$
6: $s \leftarrow task\_load[t].start$; $e \leftarrow task\_load[t].end$
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**

Task Pattern

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

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

4.1.2 Problem Description

In most graph analysis parallel programs with multiple threads (resulting from graph systems such as Galois [33] or Green-Marl [19] for example), it was observed that, each thread is in charge of groups of nodes. In that way the workload of a thread depends on 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 task pattern presented at section 4.1.1, 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 application 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.

4.1.3 Illustration Example

Figure 4.1 presents an illustration of heterogeneity of node degree. Graph (H) has fifteen nodes ($n = 15$) with an average degree of 2.4. There are three nodes with a higher degree compared to the others: node 0 with a degree of 7, nodes 1 and 2 with a degree of 6 each.

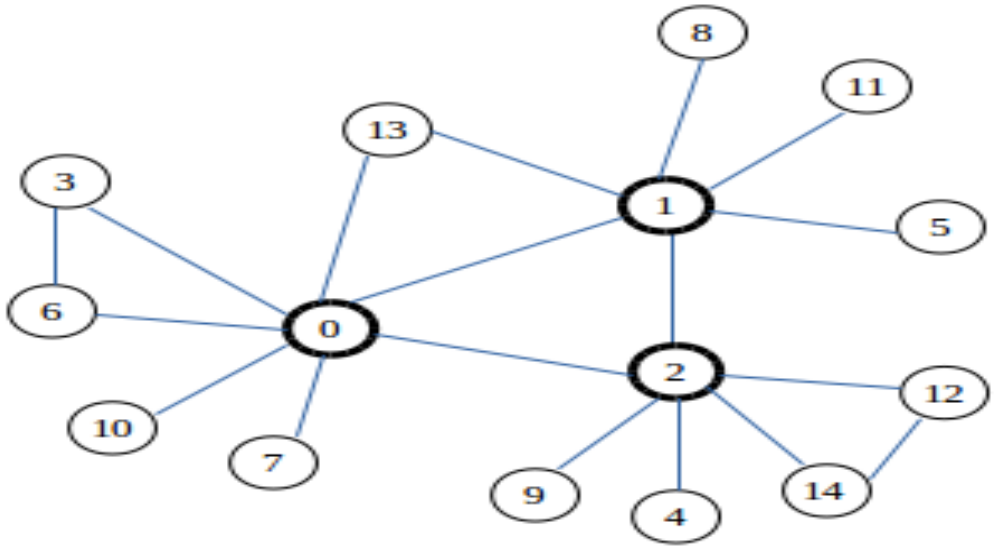


Figure 4.1: Graph (H): illustration of heterogeneity of node degree

Consider the task pattern presented in section 4.1.1. Let a_{cost} be the access cost of v at line 4. Suppose we have 3 tasks. With a proportional division of nodes among tasks, each task receives $\frac{n}{nb\_thread} = \frac{15}{3} = 5$ nodes. We assume in this example that nodes received by each task are consecutive. This leads to the following workload per task:

- Task 0 works on nodes *(0:7, 1:6, 2:6, 3:2, 4:1)* with a cost $\mathbf{22}a_{cost}$.
- Task 1 works on nodes *(5:1, 6:2, 7:1, 8:1, 9:1)* with a cost $\mathbf{6}a_{cost}$.
- Task 2 works on nodes *(10:1, 11:1, 12:2, 13:2, 14:2)* with $\mathbf{8}a_{cost}$.

The workload of task 0 is 3.66 times the workload of task 1 and 2.75 the workload of task 2. This imbalance load makes the application slower.

4.1.4 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

load balance 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** [25] 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.

4.1.5 Difference with Classical Scheduling Problem

The scheduling problem defined in this chapter is different to the classical one. In both problems, there are n threads (workers) and m tasks (work slots) with different loads. The goal is to assign tasks to workers in order to ensure load balancing among workers, i.e. to ensure that no thread will get tasks with high loads compared to other threads. This particular thread (which has tasks with high loads) will take much time compared to other threads and finally will penalize the hole multi-threaded application.

The classical problem considers that the load of each task is fixed once for all. In this case, the big deal to ensure load balancing among threads by finding the best strategy that assigns tasks to threads. A typical strategy may assign different numbers of tasks to each thread.

In our scheduling problem, the emphasis is on the tasks. This is because, in our case, tasks are carried on graphs. Contrary to the classical problem, we are able to modify the load of each task. Finally, our goal is to ensure load balancing among threads by

setting equivalent loads to each thread. By doing that, we ensure that even the simple strategy that assigns the same number of tasks to each thread will not cause imbalanced load among threads.

Figure 4.2 illustrates the difference with the classical scheduling problem. At the left, we have tasks with different loads to assign to threads in the middle. A the right, we have tasks with loads reseted to be the same. Classical problem looks for a strategy to assign tasks at the left to threads in the middle. In our problem, we are looking for a strategy to get tasks with the same load (as in the right).

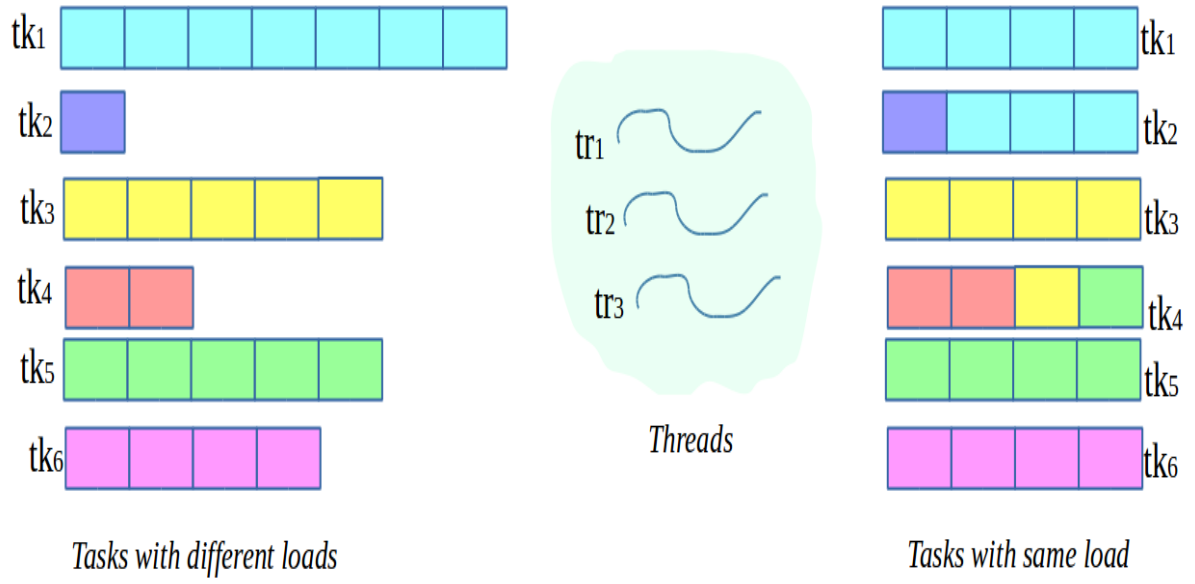


Figure 4.2: Difference with classical scheduling problem

4.2 Degree-aware Scheduling

This algorithm builds a set (or an array) 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 each task should have. The algorithm greedily adds node i to the nodes collection of task t until $\sum d_i$ reaches d_{max}. The complexity of this algorithm is O(n).

Consider the example presented at section 4.1.3; $n = 15$, $nb\_task = 3$, $d_{max} = \frac{d_{sum}}{nb_{task}} = \frac{36}{3} = 12$. Using degree-aware scheduling, we have the following workload per task:

- Task 0 works on nodes *(0:7, 1:6)* with a cost $\mathbf{13}a_{cost}$.
- Task 1 works on nodes *(2:6, 3:2, 4:1, 5:1, 6:2)* with a cost $\mathbf{12}a_{cost}$.

Algorithm 11 : **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** $task[\,]$

- Task 2 works on nodes *(7:1, 8:1, 9:1, 10:1, 11:1, 12:2, 13:2, 14:2)* with $\mathbf{11}a_{cost}$.

The workloads of tasks are now close. This allows the application to be quicker compared to the previous scheduling.

In practice, there is another parameter, chunk. The idea behind this parameter is to set the number of tasks carried by each thread. The number of task is now $nb_{task} = nb_{thread} * chunk$. Typically, if chunk = k, then each thread will receive k tasks if the scheduling is static. In the ideal case, tasks produced by our algorithm have the same cost (the same degree). In that way, even in dynamic scheduling, the number of tasks received by each thread is very close to k.

The difference between algorithm 12 and algorithm 11 is on line 1 where the number of task is set. The other lines are exactly the same.

In the next section, we present graph ordering heuristics combined to Deg-scheduling in order to ensure both cache misses reduction and load balancing among threads.

Algorithm 12 : **deg-scheduling-with-chunk** (**Deg**ree-aware **scheduling**)

Input: $G = (N, E)$,**chunk**,d_{sum}: sum of all nodes degree, nb_{thread}: number of threads

Output: $task[\,]$, a vector where each task has start and end nodes

1: $nb_{task} \leftarrow nb_{thread} * chunk$, $d_{max} \leftarrow \frac{d_{sum}}{nb_{task}}$, $t \leftarrow 0$, $i \leftarrow 0$
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** $task[\,]$

4.3 Graph Ordering and Degree Scheduling

We saw in chapter 3 that keeping nodes in good ordering allows to reduce cache misses and hence execution time. By noting 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 the design of algorithms 13, 14, 15, and 16.

4.3.1 Cn-order and Deg-scheduling

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.

Algorithm 13 : 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** $task[\,]$

4.3.2 Rabbit Order and Deg-scheduling

In algorithm 14, we first process Rabbit and then we execute degree-aware scheduling. By doing this, we ensure that nodes processed by each thread are most likely in the same sub-community and then are close in the memory.

Algorithm 14 : rabbit-deg-scheduling

Input: $G = (N, E)$,d_{sum}, nb_{thread}

Output: $task[\,]$, a vector where each task has start and end nodes

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

4.3.3 Numbaco and Deg-scheduling

In algorithm 15, the execution of numbaco ensures that nodes processed by each thread are most likely in the same community and then, as in the previous algorithm, these nodes are close in the memory.

Algorithm 15 : numbaco-deg-scheduling

Input: $G = (N, E)$,d_{sum}, nb_{thread}

Output: $task[\,]$, a vector where each task has start and end nodes

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

4.3.4 Gorder and Deg-scheduling

In algorithm 16, Gorder ensures that each thread will process sibling nodes.

4.3.5 Summary of Designed Heuristics

So far, we showed how to use complex networks properties (community structure, heterogeneity of node degree) and graph algorithm patterns to design two categories of heuristics:

Algorithm 16 : gorder-deg-scheduling

Input: $G = (N, E)$,d_{sum}, nb_{thread}

Output: $task[\,]$, a vector where each task has start and end nodes

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

- Graph ordering heuristics: Cn-order, NumBaCo, Rabbit, Gorder. The goal behind these heuristics is to increase graph application's performance by reducing cache misses.
- Degree-aware scheduling heuristics:Deg-scheduling, Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling, Gor-deg-scheduling. The goal of these heuristics is to ensure load balancing among threads (for Deg-scheduling) or to ensure both load balancing among threads and cache misses reduction.

Figure 4.3 presents the designed heuristics and knowledge used to design them. The next section shows the time complexity of theses heuristics.

4.4 Time Complexity of Designed Heuristics

As in the previous chapter, we consider a graph $G = (N, E)$, where $n = |N|$ and $m = |E|$. In Table 4.1, $cplx(H)$ is the time complexity of heuristic H. Here H can be deg-scheduling, gorder-deg-scheduling, rabbit-deg-scheduling, numbaco-deg-scheduling or comm-deg-scheduling.

H	$cplx(H)$
$deg - scheduling$	$O(n)$
$gorder - deg - scheduling$	$O(\sum_{v \in N} d_v^2 + 2n)$
$rabbit - deg - scheduling$	$O(E)$
$numbaco - deg - scheduling$	$O(n(1 + logn))$
$comm - deg - scheduling$	$O(\sum_{v \in N} d_v^2 + n(2 + logn))$

Table 4.1: Time complexity comparison

Among all the heuristics, deg-scheduling is the fastest. The complexities of the other heuristics follow the same order as the complexities of graph ordering presented in the

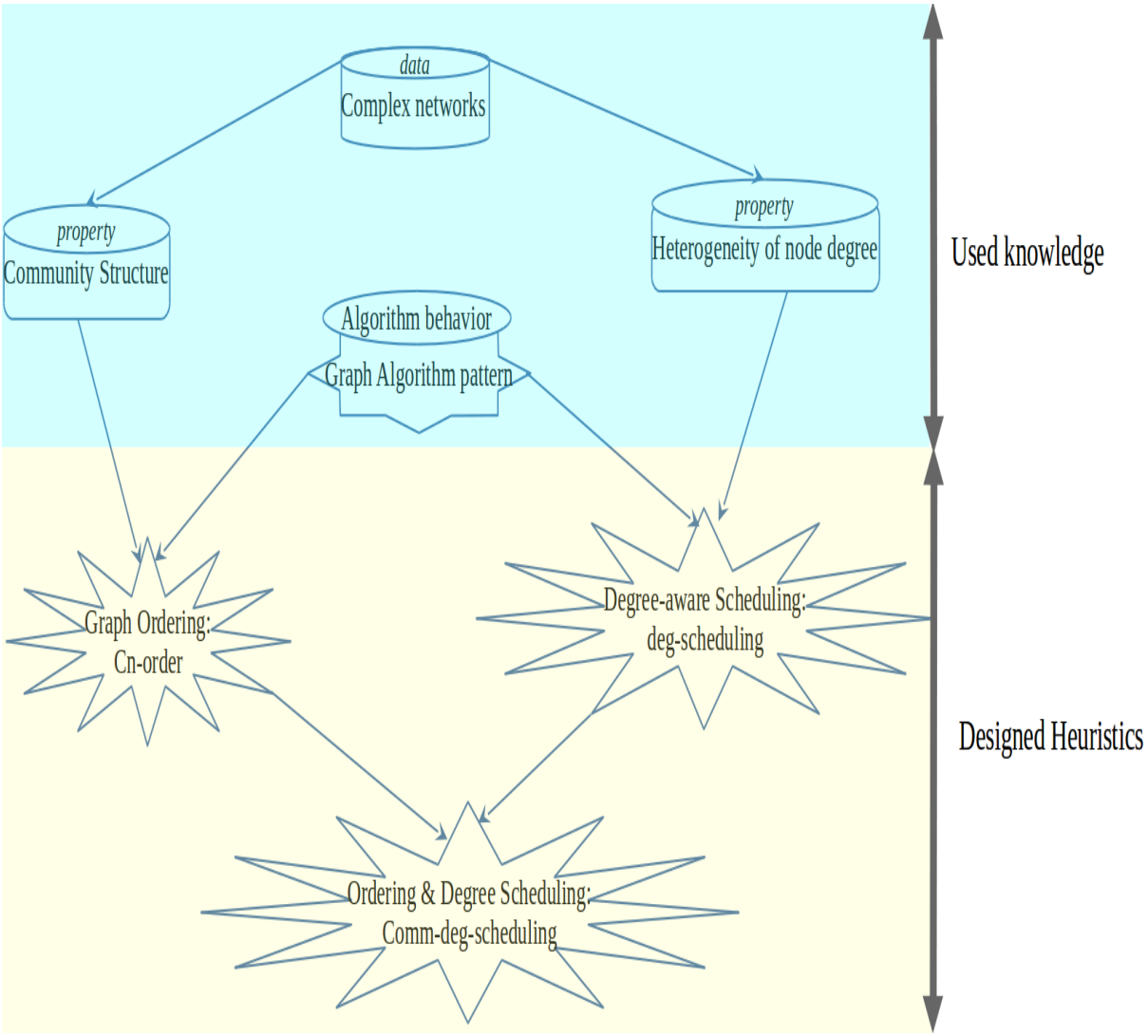


Figure 4.3: Used knowledge and designed heuristics

previous chapter (Gorder, Rabbit, NumBaCo and Cn-order). The complexities of their corresponding degree-aware scheduling is gotten by adding n to the initial complexity. In others words:

$$cplx(\text{[order]-deg-scheduling}) = cplx(\text{[order]}) + cplx(\text{deg-scheduling})$$

Where [order] is either Gorder, Rabbit, NumBaCo or Cn-order.

Table 4.1 shows that:

$$\begin{cases} cplx(\text{deg-scheduling}) < cplx(\text{rabbit-deg-scheduling}) < cplx(\text{numbaco-deg-scheduling}) \\ cplx(\text{numbaco-deg-scheduling}) < cplx(\text{gorder-deg-scheduling}) < cplx(\text{comm-deg-scheduling}) \end{cases}$$

The interpretation is also the same as in the previous chapter. With numbaco-deg-scheduling and comm-deg-scheduling for example, this observation implies that:

- $|Alg_1(numbaco - deg - scheduling)| > |Alg_1(comm - deg - scheduling)|$: the number of graph algorithms that belong to $Alg_1(numbaco - deg - scheduling)$ is greater than the number of graph algorithms that belong to $Alg_1(comm - deg - scheduling)$. In other words, there are more graph algorithms with higher time complexity compared to numbaco-deg-scheduling than graph algorithms with higher time complexity compared to comm-deg-scheduling.

- $|Alg_2(numbaco-deg-scheduling)| < |Alg_2(comm-deg-scheduling)|$: the number of graph algorithms that belong to $Alg_2(numbaco-deg-scheduling)$ is smaller than the number of graph algorithms that belong to $Alg_2(comm-deg-scheduling)$. That is, there are more graph algorithms with lesser time complexity compared to comm-deg-scheduling than graph algorithms with lesser time complexity compared to numbaco-deg-scheduling.

As with graph ordering heuristics, according to the target graph algorithm time complexity, we can always benefit from these degree-aware scheduling heuristics, either using them for preprocessing or during graph loading in program execution.

The following section presents experimental analysis of these heuristics.

4.5 Experimental evaluation

As in the previous chapter, we present results got with Pagerank [35]. We used a posix thread implementation proposed by *Nikos Katirtzis*[1]. This implementation uses adjacency list representation. But in this chapter, experiments were made with many threads on Numa4 machine described in Chapter 2 (experiments were made with only one thread in the previous chapter). Linux perf also described in Chapter 2 is used to report cache misses and cache references.

As in the previous Chapter, we used Live Journal and Orkut datasets [54]. As a reminder:

- Live Journal is an unoriented graph with 3,997,962 nodes and 34,681,189(X2) edges. The graph was represented with an adjacency list which is O(2m + n) space. The space taken by graph in memory is (2*34681189 + 3997962)*4bytes = 279.84 MB (do not fit in Numa4 L3 cache memory).

- Orkut is an unoriented graph with 3,072,441 nodes and 117,185,083(X2) edges. The graph was represented with an adjacency list which is O(2m + n) space. The space

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

taken by graph in memory is (2*117,185,083 + 3,072,441)*4bytes = 905.77 MB (do not fit in Numa4 L3 cache memory).

In the previous Chapter, results with one thread on used architectures (Core2, Numa4 and Numa24) showed that:

- All the graph orders, Gorder, Rabbit, NumBaCo and Cn-order outperform the original order.
- Gorder gives the least references to L3 compared to Rabbit and NumBaCo
- NumBaCo gives the least cache misses, compared to Rabbit and Gorder.
- Finally, Cn-order is the best among the all orders in terms of cache-references reduction, cache-misses reduction and hence execution time reduction.

In this section, we are studying how these graph orders behave with many threads. We will show that cache misses reduction and cache references reduction do not influence execution time reduction with many threads as they did with only one thread. There is another performance element that influences execution time reduction: load balancing among threads. This section does not consider $Nodes\_closeness$ because it depends only on the used graph and the target machine (when using NumBaCo or Cn-order). It stays the same regardless the number of threads.

We used cache references, cache misses, load balancing among threads and execution time reduction to compare graph ordering heuristics (Gorder, Rabbit, NumBaCo, Cn-order) and also to compare degree-aware scheduling heuristics (Gor-deg-scheduling, Rab-deg-scheduling, Numb-deg-scheduling, Comm-deg-scheduling).

4.5.1 Cache References Reduction

Figures 4.4, 4.5, 4.6, 4.7 (reported with Orkut dataset) and figures 4.8, 4.9, 4.10, 4.11 (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).

Observations

Orkut Dataset. With graph ordering heuristics in figures 4.4 and 4.5, Cn-order is the best with almost 30% of reduced cache references. It is close to Goder with almost 29%.

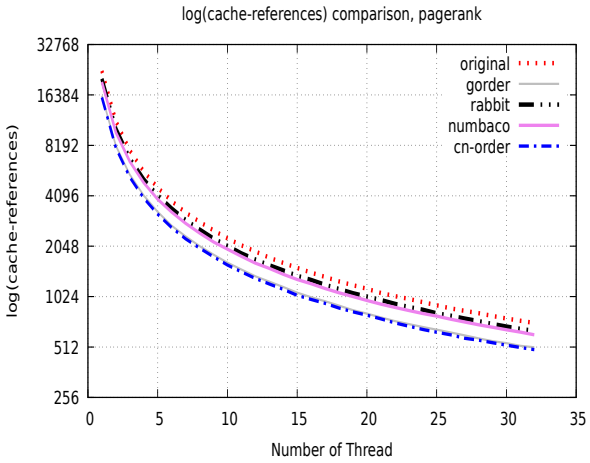


Figure 4.4: cache-ref with graph orders – Orkut

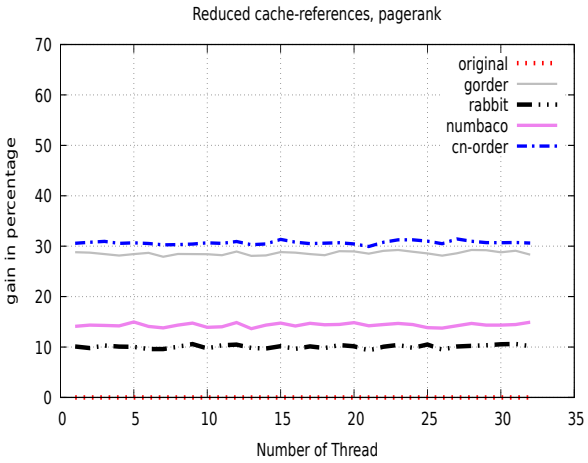


Figure 4.5: cache-ref with graph orders – Orkut

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.

With degree-aware scheduling heuristics in figures 4.6 and 4.7, Comm-deg-scheduling is the best with 30% of reduced cache references. Gor-deg-scheduling is the second with 29%, Numb-deg-scheduling is the third with almost 13% and Rab-deg-scheduling is the last with 10%.

Live Journal Dataset. In figures 4.8 and 4.9, with graph ordering heuristics, Cn-order reduces cache references more between 32% and 35%. Goder varies between 26% and 31 %. NumBaCo varies between 20% and 26%. Rabbit varies between 16% and 21%.

With degree-aware scheduling heuristics in figures 4.10 and 4.11, Comm-deg-scheduling reduces cache references more between 30% and 34%. Gor-deg-scheduling varies between 23% and 30%. Numb-deg-scheduling varies between 19% and 28%. Rab-deg-scheduling varies between 13% and 20%.

Interpretation

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

- Even with many threads, Cn-order is the best, Gorder the second, NumBaCo the third and Rabbit the last. As already explained in the previous chapter, Cn-order is the best because it uses Gorder technique that allows sibling nodes to be close in

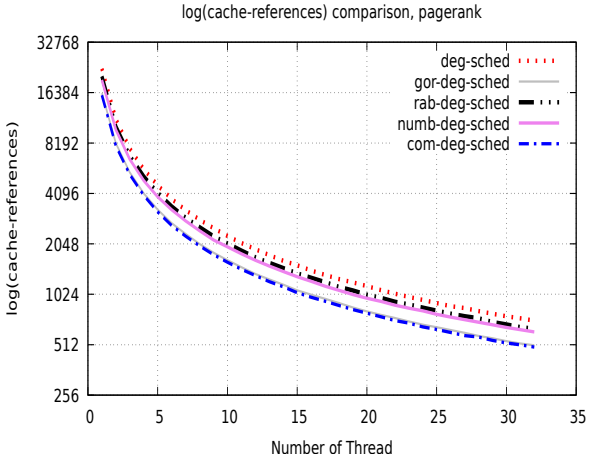


Figure 4.6: cache-ref with scheduling – Orkut

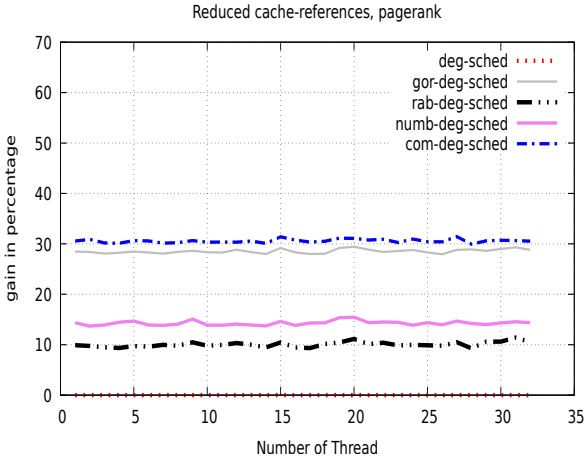


Figure 4.7: cache-ref with scheduling – Orkut

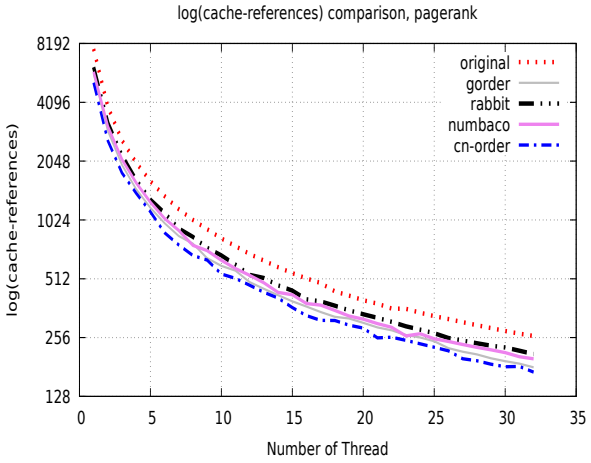


Figure 4.8: cache-ref with graph orders – Lj

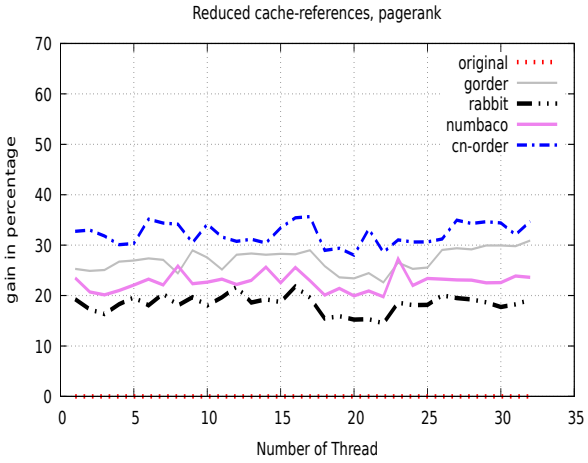


Figure 4.9: cache-ref with graph orders – Lj

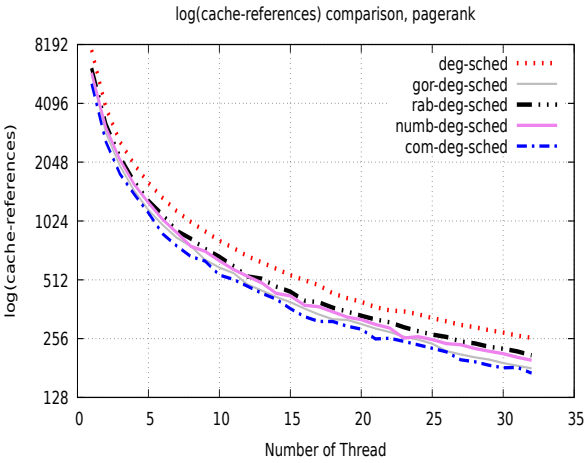


Figure 4.10: cache-ref with scheduling – Lj

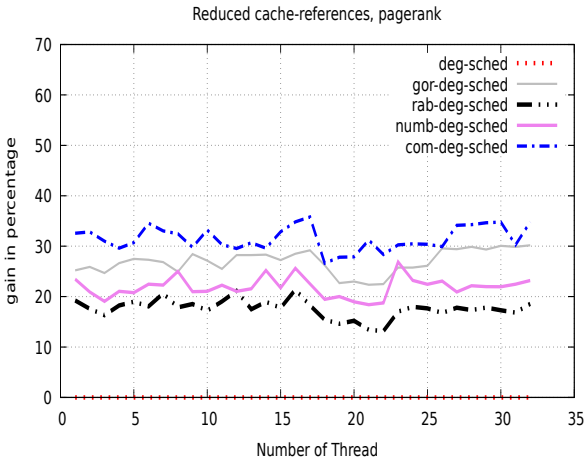


Figure 4.11: cache-ref with scheduling – Lj

memory and hence increase references to L1 and L2 (with cache hit); more L1 and L2 cache hits mean less L3 cache references.

- Degree-aware scheduling heuristics follow their homologous graph ordering heuristics: Comm-deg-scheduling is the best, Gor-deg-scheduling is the second, Numb-deg-scheduling is the their and Rabbit is the last. This means, in [ordering-degree-scheduling] heuristic, the scheduling part does not have a significant influence in cache references reduction, and the ordering part keeps its role.

Do graph ordering heuristics and degree-aware scheduling heuristics have the same behavior with cache misses reduction as with cache references reduction? The next section will study cache misses reduction on the same datasets with Pagerank and will provide an answer to this question.

4.5.2 Cache Misses Reduction

Cache misses reported with Orkut dataset are presented in figures 4.12, 4.13, 4.14, 4.15 and with Live Journal dataset in figures 4.16, 4.17, 4.18, 4.19. As with cache references reduction section, 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).

Observations

Orkut Dataset. In figures 4.12 and 4.13, Cn-order has the best cache misses reduction with 38% to 45%. From one to 19 threads, Rabbit and NumBaCo are better than Gorder

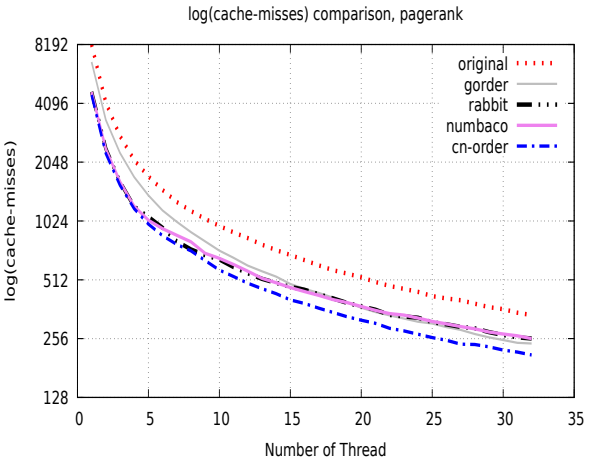


Figure 4.12: cache-misses with graph orders – Orkut

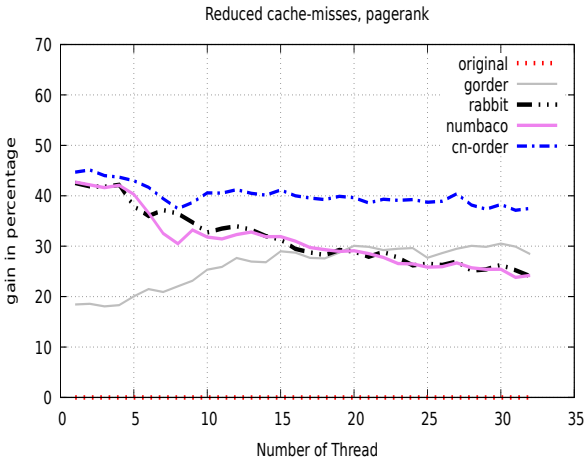


Figure 4.13: cache-misses with graph orders – Orkut

with 28% to 42% of cache misses reduction (compared to 18% – 29% with Gorder). From 20 to 32 threads, Gorder is better than Rabbit and NumBaCo with almost 30% of reduced cache misses (compared to 25% – 28% with Rabbit and NumBaCo).

In figures 4.14 and 4.15, degree-aware scheduling heuristics have the same behavior as their homologous of graph ordering heuristics. Comm-deg-scheduling is the best with 37% - 45% of reduced cache misses. From one to 19 threads, Numb-deg-scheduling and Rab-deg-scheduling (with 29% – 42%) are better than Gor-deg-scheduling (with 18% to 29%). From 20 to 32 threads Gor-deg-scheduling (with almost 30%) is better than Numb-deg-scheduling and Rab-deg-scheduling (with 24% – 29%).

Live Journal Dataset. In figures 4.16 and 4.17, Cn-order and NumBaCo have the best cache misses reduction with almost 38% to 44%. From one to 27 threads, Rabbit (with almost 22% to 35%) is better than Gorder (with 9% to 22%). From 28 to 32 threads, Gorder (23%) is better than Rabbit (22%).

In figures 4.18 and 4.19, as for Orkut dataset, degree-aware scheduling heuristics have also the same behavior as their homologous of graph ordering heuristics in cache misses reduction. Comm-deg-scheduling and Numb-deg-scheduling are the best with 30% - 45% of reduced cache misses. From one to 26 threads, Rab-deg-scheduling (with 22% – 45%) is better than Gor-deg-scheduling (with 9% to 22%). From 27 to 32 threads Gor-deg-scheduling (with almost 25%) is better than Rab-deg-scheduling (with 22%).

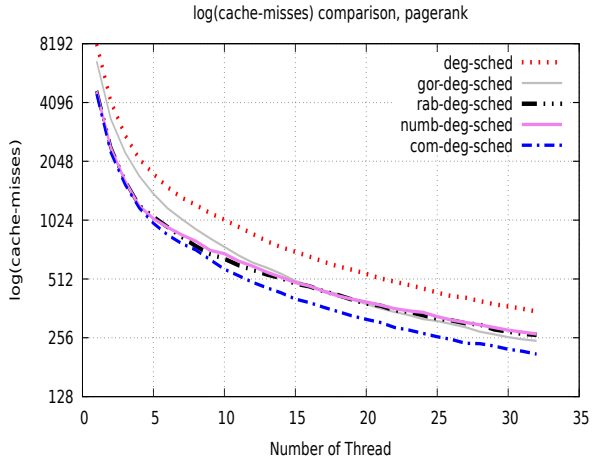


Figure 4.14: cache-misses with scheduling – Orkut

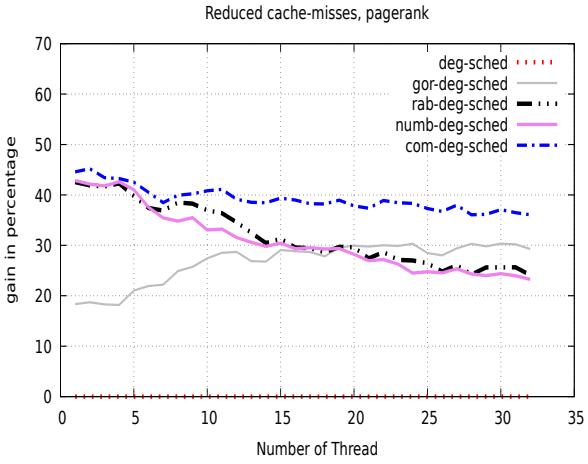


Figure 4.15: cache-misses with scheduling – Orkut

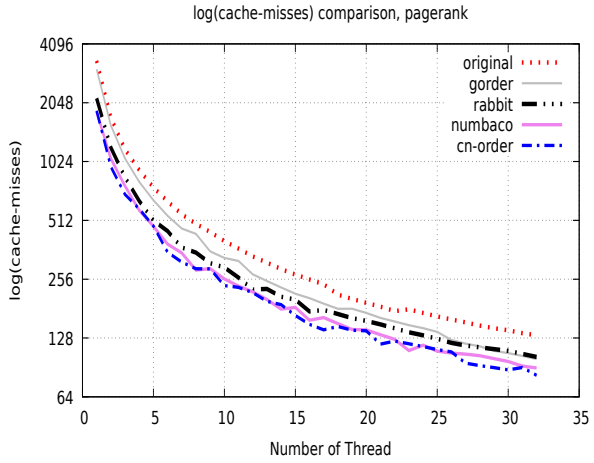


Figure 4.16: cache-misses with graph orders – Lj

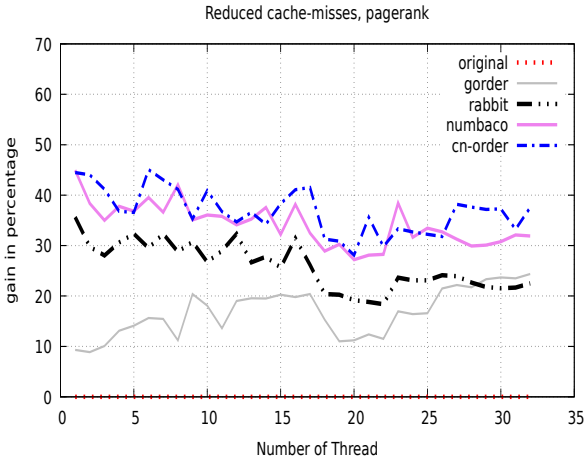


Figure 4.17: cache-misses with graph orders – Lj

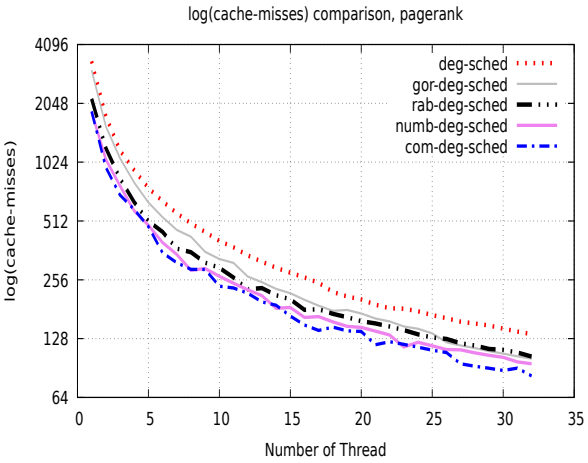


Figure 4.18: cache-misses with scheduling – Lj

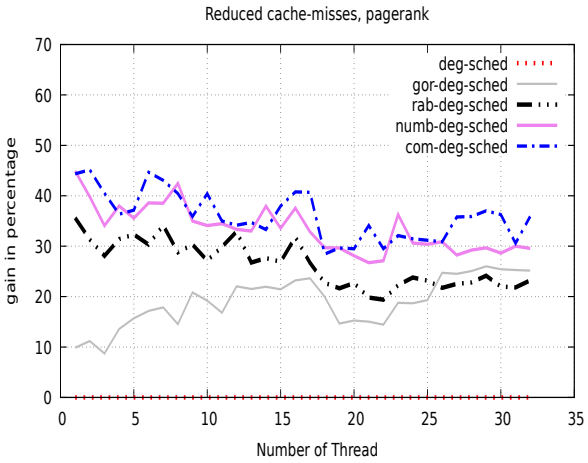


Figure 4.19: cache-misses with scheduling – Lj

Interpretation

In addition to previous observations made with Orkut and Live Journal datasets on cache misses reduction, it is also good to note that percentages of reduced cache misses are increasing form one to 32 threads with Gorder and Gor-deg-scheduling while these percentage are decreasing with the others heuristics. This can be explained by Gorder strategy and Numa4 architecture. For Numa4 architecture, it is more precisely L3 cache which is shared among groups of 8 cores: cores benefit more and more to data loaded by others. Gorder and Gor-deg-scheduling performance (cache misses reduction) benefits more to this architecture than the other heuristics, except Cn-order and comm-deg-scheduling which integrate Gorder strategy (making sibling nodes be close in memory).

- According to all the observations, we can say that, With many threads, Cn-order is always the best from one thread to 32 threads. Depending on the number of threads and the data structure:
  - With the Orkut dataset, NumBaCo and Rabbit are better than Gorder from one to one to 19 threads. And from 20 to 32, Gorder is better.
  - With the Live Journal dataset, NumBaCo is close to Cn-order. Rabbit is better than Gorder from one to one to 27 threads. And from 28 to 32, Gorder is better than Rabbit.

  The reason why there is a switch in position between Gorder curve in one side and NumBaCo and Rabbit curves in another side, is the same we invoked at the

beginning of this section: Gorder strategy and Numa4 architecture. In addition to this reason, the differences observed between the two datasets are due to the community structure of each dataset.

The switch in position starts early with Orkut (20 threads) compared to Live Journal (27 threads). This is because Orkut has very big communities (among other small communities). There are also big communities (among other small communities) in Live Journal dataset but the difference between bigger and smaller communities is not as high as in the Orkut dataset. Bigger the community compared to the cache size, less efficient are Rabbit and NumBaCo. Gorder does not depend on the community structure. Percentages of cache misses reduced with Cn-order also decreases with the number of threads because it is community structure dependent like Rabbit and NumBaCo. But these percentages stay higher than the ones with Gorder because Cn-order integrates Gorder strategy.

- As with cache references, degree-aware scheduling heuristics follow their homologous graph ordering heuristics: Comm-deg-scheduling is the best, Numb-deg-scheduling and Rab-deg-scheduling are switching their positions with Gor-deg-scheduling according to the number of threads and the dataset. As such, in [ordering-degree-scheduling] heuristic, the scheduling part does not have a significant influence in cache misses reduction, and the ordering part keeps it role.

In the next section, we study load balancing among threads.

4.5.3 Load Balancing Among Threads

Figures 4.20, 4.21, 4.22, 4.23 (gotten with Orkut dataset) and figures 4.24, 4.25, 4.26, 4.27 (gotten with Live Journal dataset) represent workload of each of the 32 threads without any heuristic (figure 4.20, 4.24), with degree-aware scheduling (figures 4.21, 4.25), with Cn-order (figures 4.22, 4.26) and with Comm-deg-scheduling (figures 4.23, 4.27). For the other heuristics (graph ordering and degree-aware scheduling), we summarize workload informations on tables 4.2, 4.3 (gotten with Orkut dataset) and tables 4.4, 4.5 (gotten with Live Journal dataset).

Orkut Dataset. In figure 4.20, each thread receives $\frac{number\ of\ nodes}{number\ of\ thread}$ nodes as its workload. Due to heterogeneity of nodes degree, this workload leads to unbalancing load. Some threads take higher time and others lesser time. For example, threads 0, 11, 12 take more than 4.7% of total time; while other threads like 30, 31, 32 take less than 2%

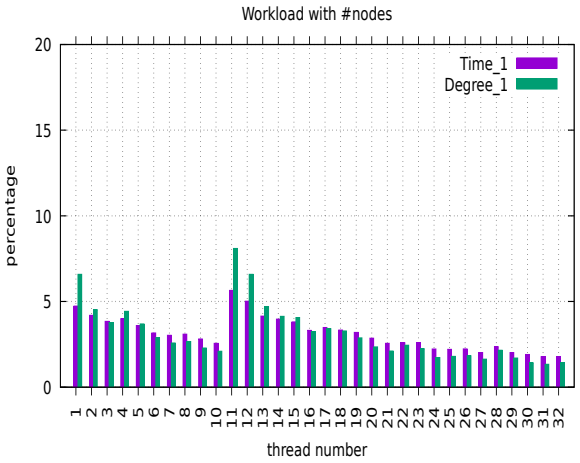


Figure 4.20: workload with node –Orkut

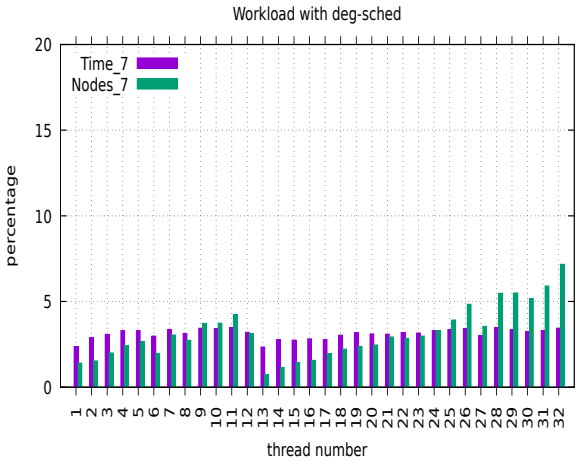


Figure 4.21: workload with deg-sched–Orkut

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 the different times is 0.97. It is interesting to note that threads with a higher time are also the one with a higher degree. This observation is used in figure 4.21 to reduce unbalancing load.

In figure 4.21, each thread receives $\frac{Total\ of\ nodes\ degree}{number\ of\ thread}$ as it workload. This workload contributes in reducing unbalancing load. Each thread now takes between 2.33% and 3.47 % of time. In this case, the standard deviation is 0.29.

In figure 4.22, we use Cn-order and the workload carries on the number of nodes. As in figure 4.20, due to the heterogeneity of nodes degree, some threads take higher time than the others. For example, thread 0 takes 11.08% of the total time (where the average one is 3.12% in case of balanced load). The standard deviation in this case is 1.71.

In figure 4.23, we use Comm-deg-scheduling to ensure load balancing among threads. The general behavior is close to what we saw at figure 4.21: each thread takes between 1.76% and 4.55% of time with a standard deviation of 0.64.

In tables 4.2 and 4.3, when comparing two by two (graph ordering heuristic vs degree-aware scheduling heuristic), we can see that the behavior of ordering heuristic is close to the one in figures 4.20 and 4.22; while the behavior of degree-aware scheduling heuristic is close to the one in figures 4.21 and 4.23.

Live Journal Dataset. In figures 4.24, 4.26 and in table 4.4, the workload is carrying on the number of nodes. As already discussed with Orkut dataset, to reduce the imbalance load due to heterogeneity of node degree, each thread receives a proportional sum of degree. The workload was reported in figures 4.25, 4.27 and table 4.5.

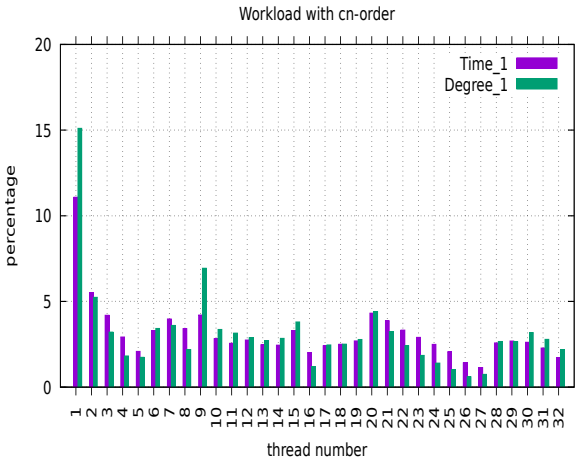


Figure 4.22: workload with cn-order –Orkut

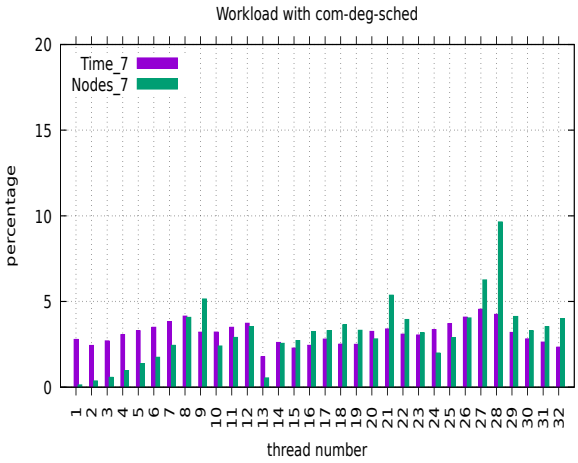


Figure 4.23: workload with com-deg-sched–Orkut

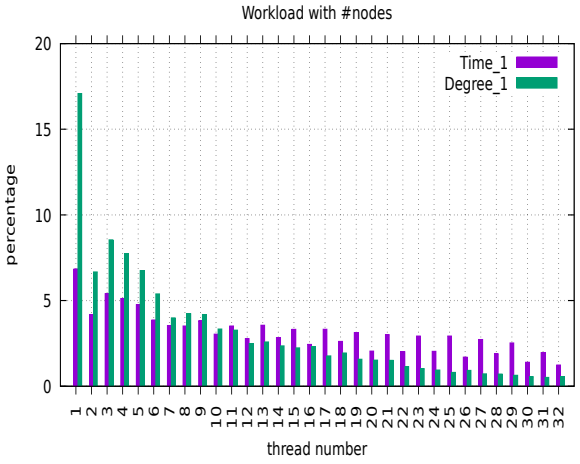


Figure 4.24: workload with node –Lj

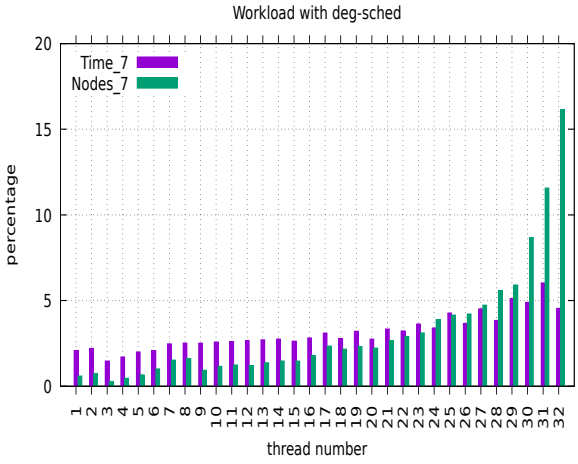


Figure 4.25: workload with deg-sched–Lj

The behavior (when comparing results with and without degree-aware scheduling) is almost the same as what we saw with Orkut dataset. For example, in figure 4.26, threads 1, 2 and 26 take more than 5% of total time; and the standard deviation of different times is 1.20. But in figure 4.27, each thread takes less than 4.53% of time and the standard deviation is 0.58.

The general observation in this section is that, the time taken by each thread depends more on the sum of degree it has than the number of nodes it has. In other words, we tend to have a balanced load when dividing the workload according to the degree than when dividing according to the number of nodes.

In the next paragraph, we will study the impact of load balancing in heuristic perfor-

Table 4.2: Workload with graph ordering heuristics(Pagerank, Orkut)

Heuristic		Original	Gorder	Rabbit	NumBaCo	**Cn-order**
Time	min (%)	1.77	0.50	1.97	1.47	1.14
	max (%)	5.64	9.84	5.84	8.21	11.08
	stdv	0.97	1.67	0.79	1.28	1.71
	var	0.95	2.80	0.62	1.65	2.93
Degree	min (%)	1.33	0.15	1.76	1.08	0.59
	max (%)	8.09	16.13	9.43	12.47	15.10
	stdv	1.63	2.99	1.55	1.99	2.52
	var	2.66	8.97	2.41	3.99	6.36
#Nodes	min (%)	3.12	3.12	3.12	3.12	3.12
	max (%)	3.12	3.12	3.12	3.12	3.12
	stdv	0.00	0.00	0.00	0.00	0.00
	var	0.00	0.00	0.00	0.00	0.00

mances (time reduction).

4.5.4 Impact in time reducing

Figures 4.28, 4.29, 4.30, 4.31 (gotten with Live Journal dataset) and figures 4.32, 4.33, 4.34, 4.35 (gotten with Orkut dataset) represent time reduction due to graph ordering heuristics (figures 4.28, 4.29, 4.32, 4.33) and degree-aware scheduling heuristics (figures 4.30, 4.31, 4.34, 4.35).

Live Journal Dataset. Remember that with one thread, Cn-order was the best in time reduction (compared to the other graph ordering heuristics). In figures 4.28 and 4.29, we can see that, except for Gorder, the other heuristic curves (Rabbit, NumBaCo, Cn-order) are switching their positions according to the number of threads. The percentage of reduced time is between 20% and 58%.

The observation of switching positions confirms that cache misses reduction and cache references reduction are not sufficient to increase the performances in multi-threaded applications. One should also take into account load balancing. This is the reason why Comm-deg-sched (in figures 4.28 and 4.29) produces the best results when it is compared to graph ordering heuristics.

In figures 4.30 and 4.31, we compare degree-aware scheduling heuristics. If Comm-deg-sched curve remains almost always the best, it switches its position (time to time)

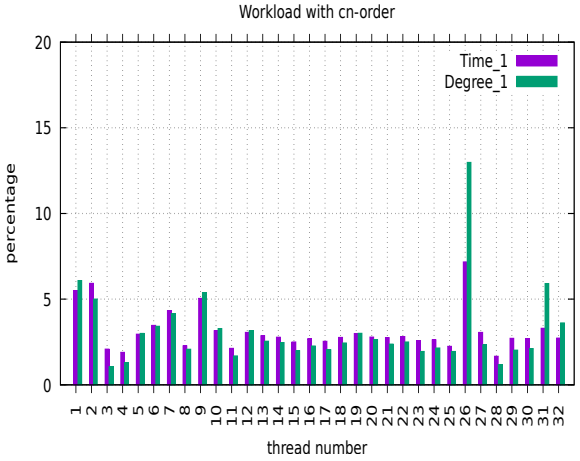


Figure 4.26: workload with cn-order –Lj

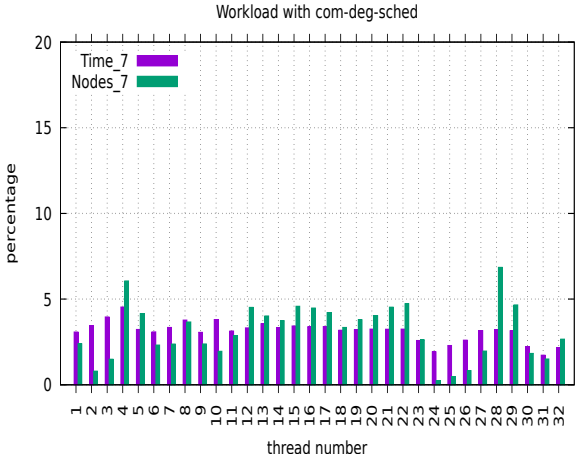


Figure 4.27: workload with com-deg-sched–Lj

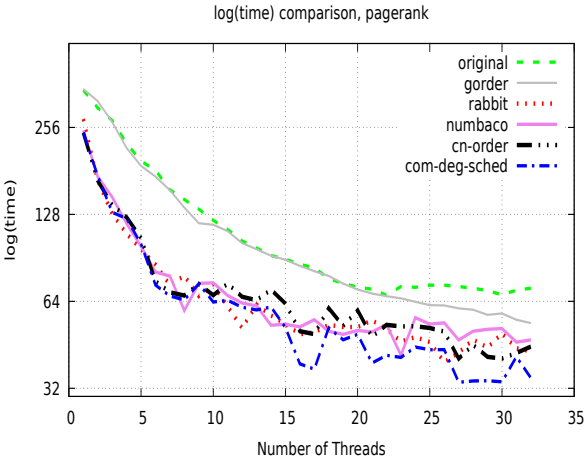


Figure 4.28: time with graph orders – Lj

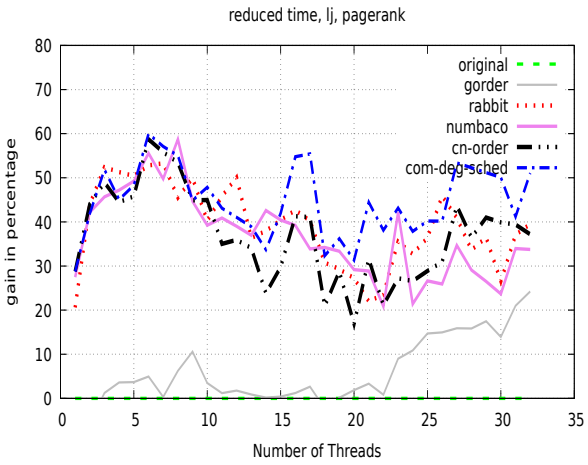


Figure 4.29: time reduced with graph orders – Lj

Table 4.3: Workload with degree-aware scheduling heuristics(Pagerank, Orkut)

Heuristic		Deg-sched	Gor-sched	Rab-sched	Numb-sched	**com-sched**
Time	min (%)	2.33	1.73	1.92	1.86	1.76
	max (%)	3.47	5.23	3.71	4.23	4.55
	stdv	0.29	0.80	0.49	0.68	0.64
	var	0.08	0.64	0.24	0.47	0.41
Degree	min (%)	3.12	3.12	3.12	3.12	3.12
	max (%)	3.12	3.12	3.12	3.12	3.12
	stdv	0.00	0.00	0.00	0.00	0.00
	var	0.00	0.00	0.00	0.00	0.00
#Nodes	min (%)	0.73	0.12	0.63	0.49	0.13
	max (%)	7.17	18.81	5.53	6.62	9.64
	stdv	1.52	3.35	1.23	1.44	1.87
	var	2.33	11.24	1.53	2.07	3.50

with the other heuristic curves. This observation suggests that load balancing is one of the most relevant performance element in multi-threaded application.

Orkut Dataset. In figures 4.32 and 4.33, we see that Cn-order produces the worst performance. The main reason for this observation is the imbalanced load. This is well explained in table 4.2 and figure 4.22: one of the thread takes 11.08% of the execution time and the standard deviation of different times is 1.71 (the highest compared to the other heuristics).

In figures 4.34 and 4.35, the behavior of heuristic curves is close to what we observed in figures 4.30 and 4.31. The reason why positions observed in cache misses reduction and cache references reduction is that load balancing is more relevant (in terms of impact in time reduction).

We showed in this section that in addition to cache misses reduction and cache references reduction, we also have another performance element: load balancing. This performance element should be used together with the other in order to increase graph analysis application efficiency. The next section will conclude the whole chapter and will give perspectives that can be followed to improve this work.

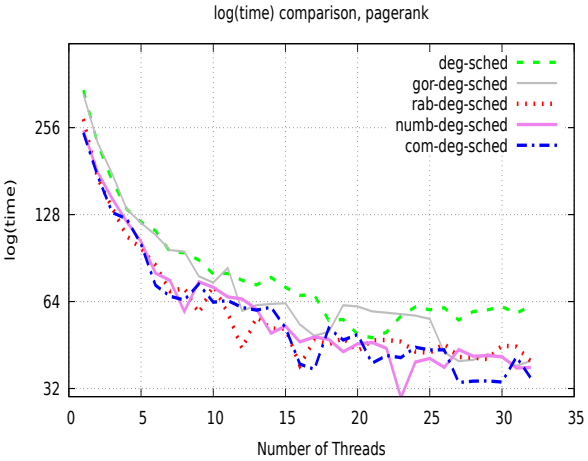


Figure 4.30: time with scheduling – Lj

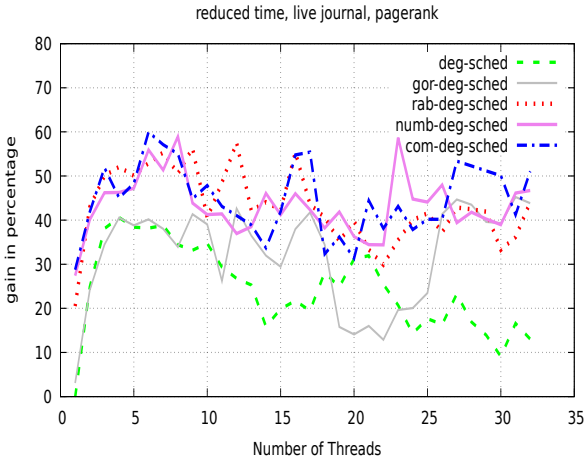


Figure 4.31: time reduced with scheduling – Lj

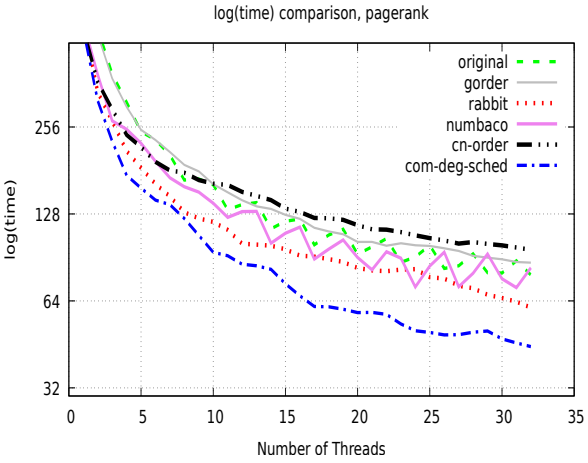


Figure 4.32: time with graph orders – Orkut

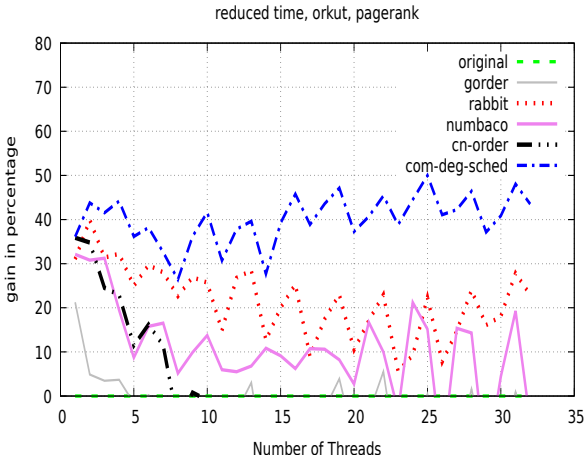


Figure 4.33: time reduced with graph orders – Orkut

Table 4.4: Workload with graph ordering heuristics(Pagerank, Live Journal)

Heuristic		Original	Gorder	Rabbit	NumBaCo	**Cn-order**
Time	min (%)	1.23	1.27	2.36	2.00	1.66
	max (%)	6.84	7.96	4.17	5.23	7.16
	stdv	1.20	1.34	0.46	0.73	1.20
	var	1.45	1.80	0.21	0.53	1.45
Degree	min (%)	0.51	0.24	2.23	1.32	1.05
	max (%)	17.07	20.98	6.58	13.10	12.98
	stdv	3.37	4.23	1.06	2.23	2.20
	var	11.41	17.90	1.14	5.00	4.85
#Nodes	min (%)	3.12	3.12	3.12	3.12	3.12
	max (%)	3.12	3.12	3.12	3.12	3.12
	stdv	0.00	0.00	0.00	0.00	0.00
	var	0.00	0.00	0.00	0.00	0.00

4.6 Discussion

Chapter Assessment. This chapter proposes Deg-scheduling, a heuristic to solve degree-aware scheduling problem, a problem formalized as multiple knapsack problem (also known as NP-complete). This chapter also proposes Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling the improved version of Deg-scheduling that use respectively Cn-order, NumBaCo, Rabbit, and Gorder to take into account graph order in scheduling.

Experimental results with many threads on Numa4 (with Pagerank and livejournal for example) showed:

- Even with multiple threads, graph ordering heuristics ensure cache misses reduction and cache references reduction. Cn-order remain the best with these two performance element.

- With many threads, cache misses reduction and cache references reduction are sufficient in time reduction, one should also take into account load balancing among thread to ensure better performance.

- Degree-aware scheduling heuristics (Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling) outperform their homologous graph ordering heuristics (Cn-order, NumBaCo, Rabbit, and Gorder) when they are com-

Table 4.5: Workload with degree-aware scheduling heuristics(Pagerank, Live Journal)

Heuristic		Deg-sched	Gor-sched	Rab-sched	Numb-sched	**com-sched**
Time	min (%)	1.46	1.74	1.75	1.40	1.72
	max (%)	6.02	5.46	3.96	4.32	4.53
	stdv	1.04	0.97	0.61	0.80	0.58
	var	1.08	0.95	0.37	0.64	0.34
Degree	min (%)	3.12	3.12	3.12	3.12	3.12
	max (%)	3.12	3.12	3.12	3.12	3.12
	stdv	0.00	0.00	0.00	0.00	0.00
	var	0.00	0.00	0.00	0.00	0.00
#Nodes	min (%)	0.26	0.17	1.40	0.26	0.23
	max (%)	16.14	25.73	4.30	6.14	6.85
	stdv	3.42	4.90	0.89	1.58	1.59
	var	11.71	24.01	0.80	2.52	2.54

pared two by two. This is because degree-aware scheduling heuristics ensure both cache misses reduction and load balancing among threads, while graph ordering heuristics ensure only cache misses reduction.

Chapter Perspectives. The work in this chapter can be improved by studying the energy footprint due to cache misses reduction, cache references reduction and load balancing among threads.

Next Chapter. In this chapter, our scheduling problem was emphasizing on tasks, i.e. we look for a strategy that allows to get tasks with the same load. In that way, we expected that, regardless the strategy to assign tasks to threads, the target graph application has great chances to be efficient (thanks to load balancing among threads).

In the next chapters, we will implement and study the designed heuristics in graph DSLs such as Galois [33] or Green-Marl [19]. These DSLs have their own strategy to assign tasks to threads. So this implementation in graphs DSLs implies that our degree-aware heuristics will be combined to DSLs scheduling strategy. Will we have any improvement due to the degree-aware heuristics?

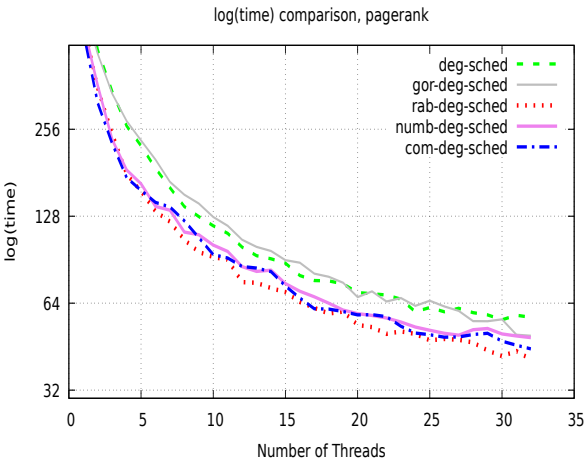


Figure 4.34: time with scheduling – Orkut

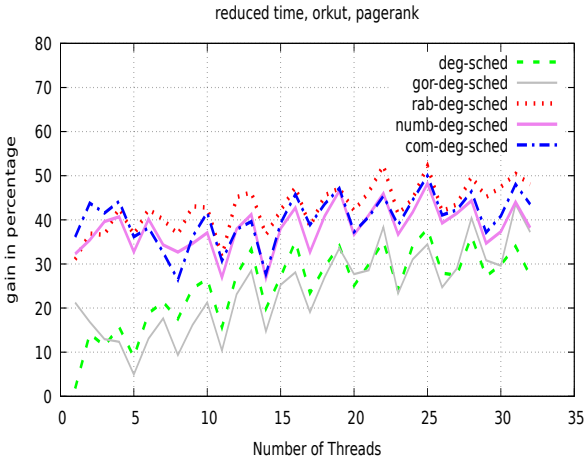


Figure 4.35: time reduced with scheduling – Orkut

Part II

Integration of Heuristics in Graph DSLs

Chapter 5

State of The Art on Graph DSLs

A Domain-Specific Language (DSL) is defined in [31] as a language designed for a precise field of applications for which it offers substantial gains of expressiveness and ease of use compared to a General Purpose Language (GPL). Another definition from [52] sees a DSL as a small (usually declarative) programming language that offers expressive power (through appropriate notations and abstractions) focused on a particular problem domain. Those definitions insist on the domain on which the DSL is designed for. In our case, the domain is graph analysis and specifically social network analysis.

There are many DSLs developed for graph analysis. But these DSLs were designed to implement classical graph algorithms, ie, with those DSLs, new algorithms from complex networks analysis do not always have an efficient implementation. This is because those existing DSLs do not exploit complex network properties to have an efficient implementation. In our quest to have such a DSL, we have two options: build a new DSL or improve existing one. Before presenting the option taken in this thesis, this chapter first revisits generalities about DSLs and then studies in detail two parallel graph DSLs: Galois and Green-Marl.

Chapter organization. This chapter starts by section 5.1 that presents patterns and guidelines used to develop DSLs. Then, the chapter presents graph DSLs in section 5.2 with details on two parallel graph analysis DSLs: Green-Marl in section 5.3 and Galois in section 5.4. The chapter ends with a discussion in section 5.5 about the choice adopted in this thesis between building a new DSL and extending an existing one.

5.1 When and How to Develop a DSL?

Developing a DSL is usually a difficult task. This is because it requires both expertise in the target domain and expertise in programming language design. Unfortunately, very few people have these two types of expertise. This section remembers the four steps in DSLs development proposed by Marjan Mernik *et al.* in [31]: 1- decision, 2- analysis, 3- conception, 4- implementation. The first step answers the question "when or in which case to develop a DSL?" and the other three steps answer the question "how to develop a DSL?". Each of these steps is associated to a set of patterns: decision patterns, analysis patterns, conception patterns and implementation patterns. Table 5.1 shows patterns used in Green-Marl and Galois developing process.

	Green-Marl	**Galois**
Decision	AVOPT	AVOPT
Analysis	Not specified	Not specified
Conception	Formal through a Grammar	Informal
Implementation	Through the compiler	C++ exploitation

Table 5.1: Patterns used by Green-Marl and Galois in developing process

Decision Patterns. For a DSL developer, these patterns are use cases that have led to the development of famous DSLs. Some of these use cases are as follows: - *Notations*, we choose between add new notations and keep existing one; -*AVOPT* that means domain Analysis, Verification, Optimization, Parallelization and Transformation; - the way to represent data structure or the way to access them.

Analysis Patterns. The goal of these patterns is to identify the problem to be solved by the DSL and collect all the necessary knowledge for that. Marjan Mernik *et al.* in [31] present three analysis patterns: -*Formal analysis*, we use well known methodologies such as Ontology-based Domain Engineering (ODE), Feature-Oriented Domain Analysis (FODA); -*Informal analysis*, there is no specific methodology; -*Existing code*, we use existing code through software tools (for example).

Conception Patterns. These patterns are grouped in two orthogonal categories. The first category concerns the relation between the new DSL and existing languages, i.e. exploitation or not of existing languages. The second category is the way to design

the DSL, i.e. formal or informal. With these two categories there are four conception patterns:

- Language exploitation. The DSL is designed totally or partially from a GPL or from another DSL. More precisely,
  - *Piggyback* is when one uses a part of an existing language.
  - Specialization is when the DSL is restricted to an existing language
  - Extension is when an existing language is extended.
- Language invention. The DSL is design from scratch.
- Formal conception. The DSL is described through formal method such as a grammar or a state machine.
- Informal conception. The DSL is described without any specific method.

In practice, formal conception is associated with language invention and informal conception is associated with language exploitation.

Implementation Patterns. There are two main approaches to implement DSLs: embedded and stand alone. In an embedded approach, a DSL is implemented in an existing GPL. In this case, the developer defines in this GPL new abstract data types and new operators. In a stand alone approach, a DSL is developed from scratch. In that case, the DSL developer can use an interpreter, a compiler or a preprocessor.

With an interpreter, a program expressed in the language is analyzed and executed instruction after instruction. An interpreter is easy to develop and extend and is used when the speed of execution is not the first goal of the language. With a compiler (or an application generator), a program is translated into different program in a lower level. With a preprocessor, DSL program elements are transformed into other elements of another language considered as the basis one; static analysis is done by the basis language. A preprocessor can appear in many forms: macro, source to source transformation, pipelining, lexical treatment.

Embedded DSLs vs Stand alone DSLs. Table 5.2 summarizes the advantages and disadvantages of each category of DSL. Further details are given in [52]. More generally, a DSL developer will choose stand alone approach in these cases: - the DSL decision is influenced by AVOPT pattern; - the DSL should have notations close to domain experts;

-the DSL is intended to have large community of developers. In other cases, it is advised to choose embedded approach.

	Stand alone DSL	**Embedded DSL**
Adv.	- Syntax is close to domain experts - Use domain specificities to do AVOPT	- Ease to develop - Reuse host infrastructures
Dis.	- Compiler or interpreter is hard to develop - There are more chances to introduce errors	- Syntax can be far to domain experts - It is hard to do AVOPT
Obs.	- Disadvantages are minimum if one reuses existing tools	- Reusing host is good, but DSL is limited

Table 5.2: Embedded DSLs vs and stand alone DSLs

When the decision to make a DSL is an evidence to the developer, he is called to follow guidelines presented in [22].

DSLs Design Guidelines. Gabor Karsai *et al.* [22] present guidelines intended to support a DSL developer in achieving better quality in his language design and a better acceptance among future users of this language. These guidelines are grouped in 5 categories:

a) *Language purpose*: this is the reason why the developer wants to design the language.

b) *Language realization*: we choose between make the language from scratch and make it from an existing language.

c) *Language Content*: these are elements to put in the language.

d) *Concrete Syntax*: this is the external representation of the language.

e) *Abstract Syntax*: this is the internal representation of the language.

More generally, these guidelines teach that the DSL developer should master the target domain and he should also have a close collaboration with experts of this domain. In our case, as a reminder, our goal is to provide DSL for parallel social network analysis.

The following sections present graph analysis DSLs with a focus on Green-Marl and Galois. We chose them because they are famous and also because it was shown that they can be used together with other DSLs.

5.2 Graphs DSLs

Graphs provide a flexible abstraction for describing relationships between discrete objects. Many practical problems in scientific computing, social science, astronomy or data analysis are modeled by graphs. These graphs are usually complex and unstructured. For that reason, early graph analysis tools like BGL [45] became quickly not enough efficient. Year after year, new graph analysis platforms are implemented (see figure 5.1). The role of the newcomers is to fill the gaps of the old ones.

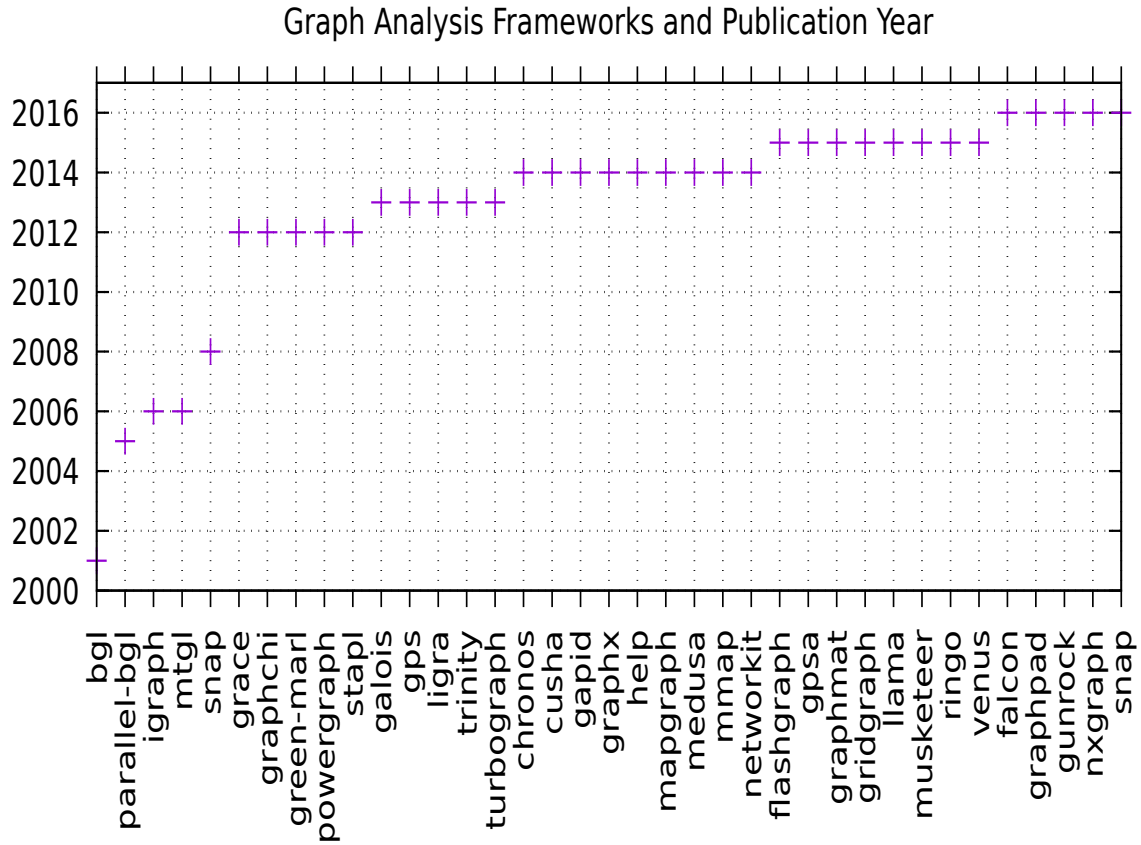


Figure 5.1: Graph Analysis Frameworks During Years

Graph analysis tools developers are usually guided by the challenges presented by Andrew Lumsdaine and co-authors [29]. The most important are: – **Capacity**, graph storage (store graph in a single machine or not); – **Performance**, how to reduce execution time of graph analysis applications? – **Implementation**, how to easily write a graph analysis program (starting with a graph algorithm)?

Graph DSLs developing is mostly motivated by performance and implementation. In

the next sections, we focus on two famous DSLs: Green-Marl [19], a stand alone DSL and Galois [33], an embedded DSL.

5.3 Green-Marl DSL

This section presents Green-Marl through a general description (its expected-usage, its limits) and its syntax (through Katz score).

5.3.1 General Description

Green-Marl is a stand alone DSL. Green-Marl was developed for parallel graph analysis. The user writes his graph analysis program in Green-Marl syntax. Then he compiles this program with Green-Marl compiler in order to get C++ code. The compiler can also generate Giraph code or GPS code. During the compiling phase, the goal is to be close to an optimized code written by hand. In C++ generated code, parallelism is ensure by including openMP directives. Graphs are represented with Yale [14] (already discussed in chapter 2).

The expected-usage of Green-Marl is presented in figure 5.2. With an initial application, the user follows these steps:

1. Extract the graph analysis part from the initial application.
2. Implement the extracted part with Green-Marl syntax.
3. Compile this part. The Green-Marl compiler performs syntactic and semantic analysis, some optimizations, and finally code generation.
4. Integrate the generated code to the initial application. Green-Marl also provides graph data structures intended to be used together with the generated code.

In order to evaluate the ease of use, we expressed in Green-Marl some graph analysis programs, for example Katz score [24]. We found that once the language syntax is mastered, it is easy to write Green-Marl programs. But programs that require complex data types are more difficult to write. This is because Green-Marl does not allow users to define their own data types.

Green-Marl Limits. We have already mentioned that Green-Marl does not allow the definition of new types. This can be fixed by modifying the Green-Marl grammar. There is another limit that should be fixed in order to make Green-Marl proper for social

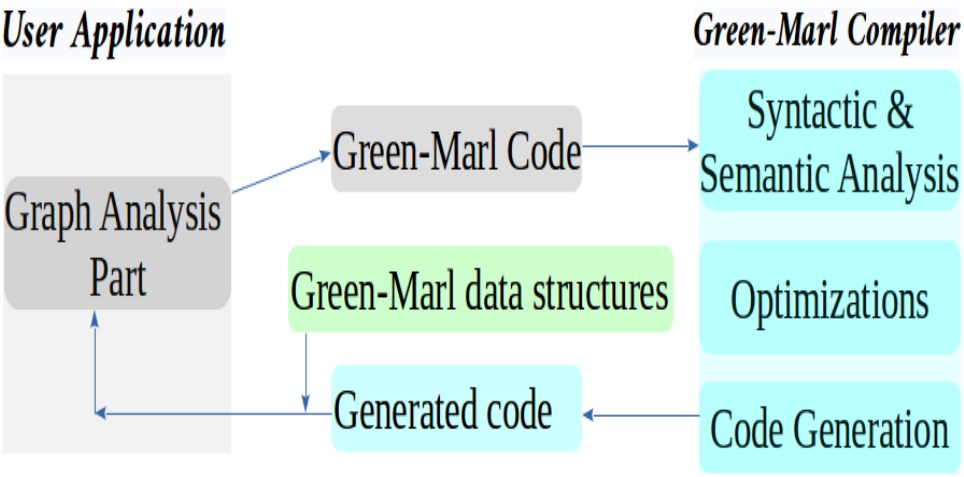


Figure 5.2: Green-Marl Usage [19]

network analysis. In fact, Green-Marl was designed for general graph. It does not take into account social graph properties such as community structure or heterogeneity of node degree. Taking into account these properties will probably increase the performance of applications generated by the Green-Marl compiler.

GPS and Giraph DSLs. The Green-Marl compiler is also intended to generate code for Graph Processing System (GPS) and Giraph. Giraph is an open source version of Pregel [30]. GPS [42] is also similar to Pregel, but with new features. Those DSLs were inspired by the bulk-synchronous parallel model [51].

The next section presents Green-Marl syntax through Katz score described at chapter 2.

5.3.2 Green-Marl Syntax with Katz Score

Before presenting Katz score written in Green-Marl, we first remember some elements of the syntax.

Green-Marl Syntax. The syntax of Green-Marl is close to that of C. But Green-Marl has specific elements such as:

- *Foreach*, *BFS*, *DFS* for loops. When Green-Marl compiler meets *Foreach* or *BFS*, it tries to generate parallel code.
- *Min*, *Max*, *Sum*, *Product*, *Count*. These are reduction instructions.

- *DGraph*, *Ugraph*, *Node*, *Edge* are Data types that can be used by the user.

- Collections: *Set*(unique and non ordered); *Order*(unique and ordered); *Sequence*(non unique and ordered).

Katz Score in Green-Marl. Lines 1 to 6 of the following code represents Katz score in Green-Marl syntax [1]. The using of *Foreach* tells the compiler to generate a parallel loop. Line 3 computes Katz score between node n and the other nodes reachable from n. The square brackets in line 3 tells the compiler to do not transform the code. We put inside these brackets a function name. This name can refer a C function or a Green-Marl procedure. In this case (this example), it refers to Green-Marl procedure presented at lines 7 to 63.

```
1Procedure katz(G:Graph,  Beta: Float, Max_l: Int):Int{
2  Foreach(n:G.Nodes){
3        [katz_score_node(G,Beta,Max_l,n);];
4  }
5  Return 0;
6}
```

We present this code in 3 parts. The first part (from line 7 to 21) is for variables declaration and initialization. We have types like *Graph*, *Node_Set*, *Node* that are specific to Green-Marl. We have also other types like *Float*, *Int*, *Map* that are close to C types.

```
7 Procedure katz_score_node(G:Graph, beta: Float, max_l: Int, x: Node){
8  Node_Set(G)neighbor_set_1,temp_set, all_reacheable_neighbor, neighbor_set_2;
9  Node_Set(G)direct_neighbor, map_path_len_1, map_path_len_2, map_temp;
10  Map<Node(G),Float> map_katz;
11  Int l = 2;
12  Int len_map_1, len_map_2, i;
13  Node key;
14  Float ktz_scr;
15  //First step, initialisation whith direct neighbors
16  Foreach(y:x.Nbrs)
17  {
18      map_path_len_1[y] = 1;
19      neighbor_set_1.Add(y);
20      direct_neighbor.Add(y);
21  }
```

The second part (from line 22 to line 57) is a Green-Marl While loop that makes the main computation of Katz score (appendix B.1). The third part (lines 58-63) is for printing results.

[1] Available at this web address: `https://github.com/messinguelethomas/modified_green_marl/blob/master/apps/src/katz_v2.gm`

```
58  //Third step, printing result
59    Foreach(y:all_reacheable_neighbor.Items){
60          ktz_scr = map_katz[y];
61          [std::cout<<$x<<" "<<$y<<" "<<$ktz_scr<<std::endl]
62    }
63}
```

Code Generated by The Green-Marl Compiler. The first Green-Marl procedure was transformed into a C function (lines 1-13). The *Foreach* loop was transformed into a parallel for loop (with OpenMP).

```
1 #include "katz_v2.h"
2 int32_t katz(gm_graph& G, float Beta, int32_t Max_l)
3 {
4    //Initializations
5    gm_rt_initialize();
6    G.freeze();
7    #pragma omp parallel for
8    for (node_t n = 0; n < G.num_nodes(); n ++)
9    {
10        katz_score_node(G,Beta,Max_l,n);
11   }
12   return 0;
13}
```

Declarations and initializations are translated by the compiler as follows (from line 14 to line 41):

```
14 void katz_score_node(gm_graph& G, float beta, int32_t max_l, node_t& x)
15{
16    //Initializations
17    gm_rt_initialize();
18    G.freeze();
19    gm_node_set neighbor_set_1(G.num_nodes());
20    gm_node_set temp_set(G.num_nodes());
21    gm_node_set all_reacheable_neighbor(G.num_nodes());
21    gm_node_set neighbor_set_2(G.num_nodes());
23    gm_node_set direct_neighbor(G.num_nodes());
24    gm_map_medium<node_t, int32_t> map_path_len_1(gm_rt_get_num_threads(),0);
25    gm_map_medium<node_t, int32_t> map_path_len_2(gm_rt_get_num_threads(),0);
26    gm_map_medium<node_t, int32_t> map_temp(gm_rt_get_num_threads(),  0);
27    gm_map_medium<node_t, float> map_katz(gm_rt_get_num_threads(),  0.0);
28    int32_t l = 0 ;
29    int32_t len_map_1 = 0 ;
30    int32_t len_map_2 = 0 ;
31    int32_t i = 0 ;
32    node_t key;
33    float ktz_scr = 0.0 ;
34    l = 2 ;
35    for (edge_t y_idx = G.begin[x];y_idx < G.begin[x+1] ; y_idx ++)
36    {
37        node_t y = G.node_idx [y_idx];
38        map_path_len_1.setValue_seq(y, 1);
39        neighbor_set_1.add_seq(y);
40        direct_neighbor.add_seq(y);
41    }
```

The main computation of Katz score generated by the compiler (from line 42 to line

89) is also available at appendix B.1. The printing results part is as follows (from 90 to line 97).

```
90 gm_node_set::seq_iter y1_I= all_reacheable_neighbor.prepare_seq_iteration();
91 while (y1_I.has_next())
92 {
93     node_t y1 = y1_I.get_next();
94     ktz_scr = map_katz.getValue(y1) ;
95     std::cout<<x<<" "<<y1<<" "<<ktz_scr<<std::endl;
96 }
97}
```

Lines 35 and 91 are a translation in C++ of *Foreach* loops. The compiler decides to do not parallelize the code. This code illustrates the fact that every *Foreach* loop or *BFS* loop is not necessary translated into a parallel loop.

In the next section, we also use Katz score in order to describe Galois syntax.

5.4 Galois DSLs

As in the previous section with Green-Marl, this section presents Galois through a general description and its syntax (with Katz score).

5.4.1 General Description

The Galois system is an implementation of the Amorphous Data-Parallelism (ADP) concept. It is a data-centric programming model designed to facilitate parallelization of both regular and irregular applications [37]. In order to write his programs, Galois user has at his disposal a set of C++ templates and data structures.

Graphs use Yale [14] representation as in Green-Marl. Galois has an internal scheduler that handles tasks and threads during execution. This is the reason why Galois has better performance than other DSLs (for example Ligra [44] or Powergraph [18]). When comparing Galois and Green-Marl performances with Katz's score on Numa24 (described in chapter 3), we found that the execution time of Green-Marl was about 6 to 4 times larger than that of Galois.

For ease of use, in comparison to Green-Marl (when the syntax is already well known), we can say that it is harder to program with Galois than with Green-Marl. In fact, the Galois developer should prepare his code before filling out the C++ templates offered to him. He should specify the part of his code than will be executed in parallel. But with Galois, the user can easily add new data types.

Galois Limits The main strength of Galois is its internal scheduler that allows it to have better performance than other DSLs. But Galois authors have not made any effort to ensure ease of programming. Another difficulty in Galois is its limited documentation. This makes Galois hard to extend.

As in the case of Green-Marl, Galois was designed for Graph general algorithms. In that way, there is no specific mechanism that exploit social graph properties in order to improve performance.

Ligra DSL Ligra is a **li**ghtweight **gra**ph processing framework written in C and designed to make graph traversal easy to write. The main functions of Ligra are VertexMap (for mapping over vertices) and EdgeMap (for mapping over edges). It was shown in [33] that implementing Ligra API at the top of Galois produces better performance (compared to Ligra alone).

5.4.2 Galois Syntax with Katz Score

As in the case of Green-Marl, we first remember some elements of the syntax before presenting Katz score written in Galois.

Galois Syntax. In order to facilitate the parallelization of graph applications, the Galois system offers a set of C++ templates (*for_ each* constructs for example) and data structures (Graph data structures for example). A parallel loop (written with *for_ each* or other construct) has an object function that represents loop body.

Katz Score in Galois. This paragraph illustrates Galois syntax with Katz score. There are 3 principal parts:

- The first part is the function that computes the Katz score between a node x_{id} and its indirect neighbors. This function is fully detailed at appendix B.2.

```
void katz_score_node(Graph& G, float beta, int32_t max_l, int32_t x_id)
{
  /*Code to compute Katz score between x_id and all reachable nodes*/
}
```

- The second part describes *for_ each* loop body; i.e. the instruction that will be executed when *for_ each* is mentioned in the main. At line 3, we see that it is the function described previously that is called (*katz_ score_ node*).

```
1 struct Katz_score {
2  void operator()(GNode& n, Galois::UserContext<GNode>& ctx) const {
3     katz_score_node(graph, beta,  max_l, graph.getData(n));
4  }
5 };
```

- The last part contains *for_each* instruction at line 9. The loop body function *Katz_score()* is referenced in this instruction. With line 9, all the threads in the application will run the graph (node by node) in parallel. For each node, function *Katz_score()* will be executed.

```
6 int main(int argc, char **argv) {
7   LonestarStart(argc, argv, 0,0,0);
8   node_vect = load_graph_gml(argv[1], &graph);
9   Galois::for_each(graph.begin(), graph.end(), Katz_score());
10  return 0;
11}
```

The next section concludes this chapter.

5.5 Discussion: Design or Extend, What to Do?

This chapter presented generalities about DSLs: what is a DSL, in which case it is advised to develop a DSL. This chapter also presented Graph DSLs with an emphasis in Galois and Green-Marl. In our quest to provide a DSL for efficient social network analysis, this chapter gives us two options:

- Design a new DSL.
- Or extend an existing one.

In this thesis, we choose the second option mainly because it requires less development time.

The next chapter shows the implementation in DSLs of heuristics that was presented in chapters 3 and 4 (graph ordering heuristics and degree-aware scheduling heuristics). This chapter will show particularly if we have any improvement in Galois and Green-Marl due to these heuristics.

Chapter 6

Graph DSLs Improvement

Instead of building a new parallel graph analysis DSL, we decide to improve the existing ones by making them be proper to complex network analysis domain. The final goal is to make efficient the implementation of algorithms from complex network analysis. Our strategy is to integrate heuristics designed and presented at chapter 3 and chapter 4 into that DSLs. This strategy requires to first provide a general methodology for heuristics integration in parallel graph analysis DSLs and then use this methodology in existing DSLs.

Chapter organization. The remainder of this chapter starts with section 6.1 that presents a general methodology about heuristics integration in DSLs. Then, section 6.2 presents methodology of heuristics integration in Green-Marl DSL. Section 6.3 presents methodology of heuristics integration in Galois DSL. Experimental results of these integrations are presented in section 6.4 with cache references reduction, in section 6.5 with cache misses reduction and in section 6.6 with time reducing impact. And finally, section 6.7 gives the synthesis of this chapter and the direct perspectives.

6.1 Methodology of Heuristics Integration On DSLs

This section presents a general methodology for heuristics integration on DSLs. We start first with graph ordering heuristics integration methodology at section 6.1.1. Then we show at section 6.1.2 the methodology of degree-aware scheduling heuristics integration.

6.1.1 Graph Ordering Heuristics Integration

As presented in chapter 3, these are: algorithm 2 (*gorder*), algorithm 3 (*rabbit*), algorithm 6 (*numbaco*), algorithm 7 (*cn-order*). We showed that, in order to increase memory locality, all these heuristics should be used before running target graph algorithms. We also showed that the target graph algorithm time complexity determines the best way to use heuristics, i.e. either for graph preprocessing or during graph loading in program execution (see section 3.6). This allows to define two case of graph ordering heuristics integration:

- **Use these heuristics for graph preprocessing.** In this case, graph ordering heuristics can stay out of the DSL. The only effort to make is to ensure that, the preprocessed graph format is compatible with the DSL graph format. Figure 6.1 illustrates this first case.

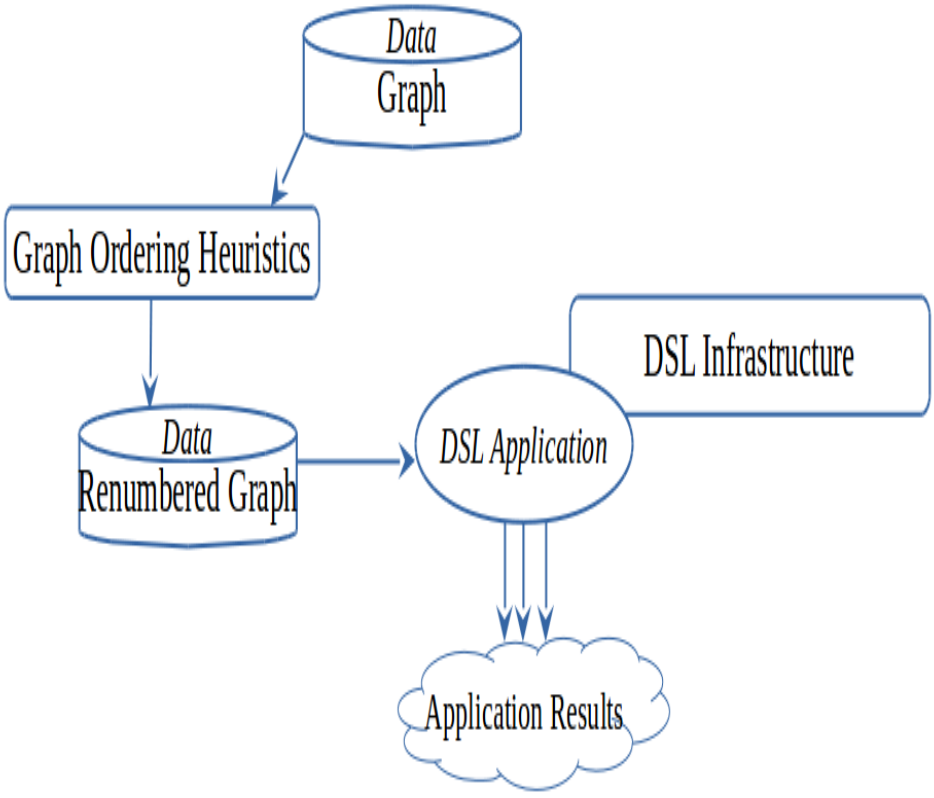


Figure 6.1: Graph Ordering Heuristics Integration (out of DSL)

- **Use these heuristics during graph loading in program execution.** In this case, graph ordering heuristics should be integrated in the DSL code. For that reason, one should first locate graph loading functions part in the DSL. Then,

this part should be modified by adding functions that implement graph ordering heuristics. In that way, every time a DSL application loads a graph, one of the graph ordering heuristics will be automatically executed and the resulting graph will be well stored in the memory. This case is illustrated in figure 6.2.

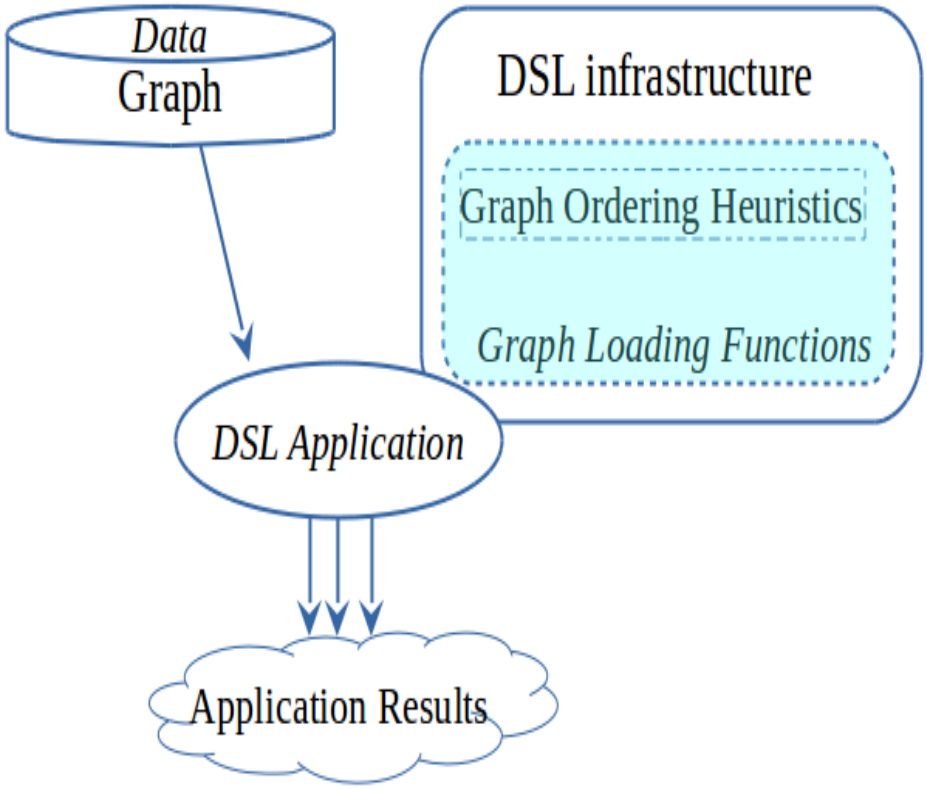


Figure 6.2: Graph Ordering Heuristics Integration (inside DSL)

6.1.2 Degree-aware Scheduling Heuristics Integration

Chapter 4 presented degree-aware scheduling heuristics: algorithm 12 (*deg-sched*), algorithm 13 (*com-deg-sched*), algorithm 14 (*rab-deg-sched*), algorithm 15 (*numb-deg-sched*), algorithm 16 (*gor-deg-sched*). These heuristics were designed to take into account both cache misses reduction and load balancing among threads. This chapter showed that these heuristics were gotten by a combination of Deg-scheduling and graph ordering heuristics. This way of seeing makes it possible to propose a two-step methodology to implement degree-aware scheduling heuristics in DSL:

- Implement graph ordering heuristics in DSL (explained in section 6.1.1).
- Implement Deg-scheduling in DSL.

In order to implement Deg-scheduling in DSL, one should first locate the place where the DSL ensures thread management. In many DSLs, it is usually located in functions related to loops on graph nodes. In DSLs with a compiler (like Green-Marl), this implementation is done at compiler level. It is then available for every application generated by the compiler. In other DSLs without a compiler like Galois, this implementation is done for each graph application.

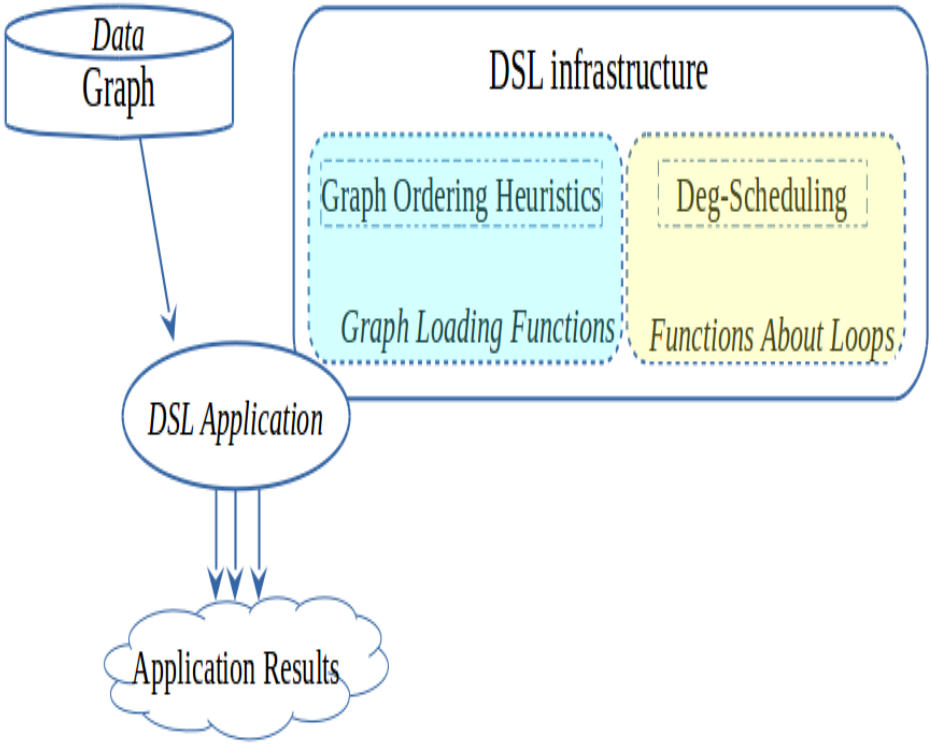


Figure 6.3: Degree-Aware Scheduling Integration

The next section presents the implementation of heuristics in Green-Marl.

6.2 Heuristics Implementation On Green-Marl

We saw in the previous chapter that Green-Marl is a stand-alone DSL. Its compiler takes a program written in Green-Marl syntax and generates a C++ code. The compiler ensures parallelism in the generated code by including openMP directives. The openMP version considered by the compiler is < 3.1.

In this section we show modifications done in Green-Marl compiler in order to implement graph ordering heuristics and degree-aware scheduling heuristics. We illustrate

these modifications with Preferential Attachment algorithm (presented at chapter 2).

6.2.1 Modifying Green-Marl Compiler

The implementation we did makes a distinction on openMP versions:

- We consider like old openMP version, the initial one supported by Green-Marl compiler. That is a version < 3.1.
- The new version for us is the one that supports *taskloop construct*. That is version > 4.5.

The idea behind *taskloop construct* is to allow dividing iterations of a loop into tasks. Due to irregularity of graph applications, this construct is the most appropriate in many cases compared to *parallel loop construct* (which is using in Green-Marl).

As indicated in the general methodology, the first thing to do is to look for functions related to graph loading and functions about loops. In the case of Green-Marl:

- Functions related to graph loading are located in the folder **apps/output_cpp/gm_graph**. In this folder, we modified two files
  - **src/graph_gen.cc** that contains functions codes;
  - **inc/graph_gen.h** that contains functions headers.
- Functions related to loop management are located in the compiler back-end.
  - In the file **src/backend_cpp/gm_cpp_gen.cc**, there is a function used to prepare parallel for construct. We use this function to choose between *taskloop construct* and *parallel loop construct*.
  - There is a function used to generate foreach header located at file **src/backend_cpp/gm_cpp_gen_foreach.cc**. We change iterators limits through this function.

Graph Ordering Heuristics Integration. These heuristics are implemented in Green-Marl through loading functions (consult this link[1]). As discussed in section 6.1.1, we have two choices:

[1] `https://github.com/messinguelethomas/modified_green_marl/blob/master/apps/output_cpp/gm_graph/src/graph_gen.ccl`

- Either graph ordering heuristics are executed outside Green-Marl and then functions in **src/graph_gen.cc** files are used to load the graph with a proper numbering.
- Or the C++ codes of these heuristics are associated to functions in **src/graph_gen.cc** file. In that case, the used graph is renumbered while it is loaded.

Degree-aware Scheduling Heuristics integration. These heuristics are implemented in Green-Marl through the function *generate_task_index* (located at the previous mentioned link). This function generates an array of tasks that will be assigned to threads. The array of tasks is represented by $G.task\_tab[\ ]$ and each task tsk contains consecutive nodes from $G.task\_tab[tsk].start$ to $G.task\_tab[tsk].end$. The iterators limits are changed in file **src/backend_cpp/gm_cpp_gen_foreach.cc** in order to make threads iterate in these tasks during the execution.

Note that other modifications that are not described in this thesis was done. We presented only the must important ones. For example making *taskloop construct* being available requires some modifications in Green-marl compiler source code. In fact, changing openMP version implies changing gcc version and finally that implies some modifications of source code.

All the modifications we did are available at github[2] like early Green-Marl compiler[3]. In the next sections, we illustrate these modifications with Preferential Attachment.

6.2.2 Preferential Attachment with Green-Marl Syntax

Preferential Attachment algorithm was described in chapter 2. The following code (registered in the file preferential_attachment.gm) represents this algorithm written with Green-Marl syntax.

```
1 Procedure preferential_attachment(G:Graph){
2 Foreach(n1:G.Nodes){
3   Foreach(n2:G.Nodes)(n1 < n2){
4   Int pref_attach = n1.Degree() * n2.Degree();
5   [std::cout<<$n1<<" "<<$n2<<" "<<pref_attach<<std::endl];
6 }}}
```

We have two *Foreach* loops that operate on graph nodes. The innermost *Foreach* has a condition that allows to eliminate redundant work. Without this condition, program will

[2] https://github.com/messinguelethomas/modified_green_marl

[3] https://github.com/stanford-ppl/Green-Marl

print Preferential Attachment for both (x_1, x_2) and (x_2, x_1). Since the graph is assumed to be undirected, these couples of nodes have the same Preferential Attachment score.

Code in square brackets will be left as it is during the compilation time. In the next section we will show the c++ code that was generated by Green-Marl compiler from this program.

In this code, we decide to write loops with two *Foreach*. One may decide to change the innermost *Foreach* (line 3) loop into *For* loop. This change will lead the compiler to generate a different parallel code as we will see in the following paragraph.

```
3   For(n2:G.Nodes)(n1 < n2){
```

6.2.3 Preferential Attachment: Generated Code

Here is code generated by Green-Marl compiler. The file refereed at first line contains functions headers added by the compiler. The user can modify this file. Lines 3 and 4 are Initialization functions invoked by Green-Marl compiler. These functions are implemented in the compiler back-end. Lines 6 to 13 are the translation in C++ of *Foreach* loops. Green-Marl compiler decided to parallelize the innermost loop by adding a *parallel loop* construct. The decision to parallelize the innermost loop is based on the assumption that the graph instance is large enough to consume all the processor and memory bandwidth of the given system. The *parallel loop construct* can be either dynamic or static.

Green-Marl compiler generates a static parallel for loop if the *Foreach* loop body does not contain *For* or *While* inside. Line 6 in the following code is a static parallel for. This is because the innermost *Foreach* loop body presented at section 6.2.2 does not contain *For* or *While*.

```
1 #include "preferential_attachment.h"
2 void preferential_attachment(gm_graph& G){
3 gm_rt_initialize(); //Initializations
4 G.freeze();//Initializations
5 for(node_t n1 = 0; n1 < G.num_nodes(); n1++){
6 #pragma omp parallel for
7 for(node_t n2 = 0; n2 < G.num_nodes(); n2++){
8  if(n1 < n2){
9    int32_t pref_attach = 0 ;
10   pref_attach = (G.begin[n1+1] - G.begin[n1]) * (G.begin[n2+1] - G.begin[n2]);
11   std::cout<<n1<<" "<<n2<<" "<<pref_attach<<std::endl;
12}}}}
```

The compiler will generate a dynamic parallel for loop if the loop body contains *For* or *While* inside. In this case, the chunk is fixed to 128. For Preferential Attachment example, if we had initially a *For* loop instead of the innermost *Foreach* (see section

6.2.2), the compiler would parallelize the outermost for loop with a dynamic schedule (line 5):

```
1 #include "preferential_attachment.h"
2 void preferential_attachment(gm_graph& G){
3 gm_rt_initialize(); //Initializations
4 G.freeze();//Initializations
5 #pragma omp parallel for schedule(dynamic,128)
6 for(node_t n1 = 0; n1 < G.num_nodes(); n1++){
7 for(node_t n2 = 0; n2 < G.num_nodes(); n2++){
8  if(n1 < n2){
9    int32_t pref_attach = 0;
10   pref_attach = (G.begin[n1+1] - G.begin[n1]) * (G.begin[n2+1] - G.begin[n2]);
11   std::cout<<n1<<" "<<n2<<" "<<pref_attach<<std::endl;
12}}}}
```

This code is the one gotten without modifications we did in Green-Marl compiler. The next section shows changes induced by our modifications. As already said, we have two cases: - with old openMP, - with new openMP.

6.2.4 Preferential Attachment With Old OpenMP

The difference between this code and the previous is the way tasks (set of nodes) are assigned to each thread and also the granularity of these tasks (lines 5-6 and 8-9). At section 6.2.3 each thread (of the parallel innermost loop) receives nodes one by one. But now, each thread receives nodes bloc by bloc.

```
5 for(int tsk_n1 = 0; tsk_n1 < G.num_task(); tsk_n1++)
6 for(node_t n1 = G.task_tab[tsk_n1].start; n1 < G.task_tab[tsk_n1].end; n1++){
7 #pragma omp parallel for
8 for(int tsk_n2 = 0; tsk_n2 < G.num_task(); tsk_n2++)
9 for(node_t n2 = G.task_tab[tsk_n2].start; n2 < G.task_tab[tsk_n2].end; n2++){
10 if(n1 < n2){
11   int32_t pref_attach = 0 ;
12   pref_attach = (G.begin[n1+1] - G.begin[n1]) * (G.begin[n2+1] - G.begin[n2]);
13   std::cout<<n1<<" "<<n2<<" "<<pref_attach<<std::endl;
14 }}}}
```

Remember that the output of the scheduling algorithm (presented at chapter 4) is an array of tasks that will be assigned to threads. Each task is a set of consecutive nodes. In this code, the array of tasks is represented by $G.task\_tab[\ ]$ and each task tsk contains consecutive nodes from $G.task\_tab[tsk].start$ to $G.task\_tab[tsk].end$

6.2.5 Preferential Attachment With New OpenMP

The difference between the following code and the one at section 6.2.4 is that *parallel loop construct* was replaced by *taskloop construct* (lines 7-9). In that way, the generated code benefits to new features offered by this construct and available in recent version of

openMP (>4.5).

```
5 for (int tsk_n1 = 0; tsk_n1 < G.num_task(); tsk_n1++)
6 for (node_t n1 = G.task_tab[tsk_n1].start; n1 < G.task_tab[tsk_n1].end; n1++){
7 #pragma omp parallel
8 #pragma omp single
9 #pragma omp taskloop
10for (int tsk_n2 = 0; tsk_n2 < G.num_task(); tsk_n2++)
11for (node_t n2 = G.task_tab[tsk_n2].start; n2 < G.task_tab[tsk_n2].end; n2++){
12if (n1 < n2){
13  int32_t pref_attach = 0 ;
14  pref_attach = (G.begin[n1+1] - G.begin[n1]) * (G.begin[n2+1] - G.begin[n2]);
15  std::cout<<n1<<" "<<n2<<" "<<pref_attach<<std::endl;
16}}}}
```

6.3 Heuristics Implementation in Galois

We presented Galois system in the previous chapter as an embedded DSL. It offers to user a set of c ++ templates and data structures. These templates are used to facilitate parallelization of both regular and irregular applications. A program in Galois should follow this:

- Loops (run in parallel) must be written using Foreach constructs.
- Each For_each must use one of the Galois-provided work-list classes.
- Data structures (accessed in parallel) are expressed in Galois-provided classes.

This section shows modifications done in Galois in order to implement graph ordering heuristics and degree-aware scheduling heuristics. As with Green-Marl case, we illustrate these modifications with Preferential Attachment algorithm (presented at chapter 2).

6.3.1 Heuristics implementation

With Galois, there is no compiler. Thus, every modification due to heuristics implementation are done for each graph analysis application. In other words, the process is repeated every time we change the graph analysis application. But there is a general approach to do it: a parallel loop in Galois has an object function that represents loop body, heuristics are implemented in this function.

In the following sections, we will show how heuristic implementation is done with Preferential Attachment score.

6.3.2 Preferential Attachment in Galois

The following code represents Preferential Attachment in Galois without heuristics. There are 3 parts:

- Core function of Preferential Attachment score. It will be executed by each thread.

```
void preferential_attach_node(Graph& G, int32_t n1){
int nb_node = node_vect.size(), n2, attach_scr;
for(n2 = 0; n2 < nb_node; n2++){
if(n1<n2){
        attach_scr = nb_neighbors(n1, G)*nb_neighbors(n2, G);
        std::cout<<n1<<" "<<n2<<" "<<attach_scr<<std::endl;
}}}
```

- Loop body that calls the core function.

```
struct preferential_attach{
 void operator()(GNode& n, Galois::UserContext<GNode>& ctx) const {
  preferential_attach_node(graph, graph.getData(n));
}};
```

- Main contains a for_each loop that launches the parallel execution.

```
int main(int argc, char **argv) {
2 int nb_node = atoi(argv[2]);
3 node_vect = load_graph_txt(argv[1], &graph,nb_node);
4 Galois::for_each(graph.begin(), graph.end(), preferential_attach());
5 return 0;
6}
```

In the next section we show changes on this code that happen with heuristics.

6.3.3 Preferential Attachment in Galois with Heuristics

The first part (core function) is left without any change. Changes are done in loop body and main parts.

```
struct pref_attach_deg{
  void operator()(GNode& node, Galois::UserContext<GNode>& ctx) const {
   int32_t n = graph_for_task.getData(node);
   for(int i = task_tab[n].start; i < task_tab[n].end; i++)
      preferential_attach_node(graph, graph.getData(node_vect.at(i)));
}};
```

In the loop body part, we change the granularity. Instead of working with only one node, each thread will work on a bloc of nodes (refereed as tasks). As explained in degree scheduling algorithm presented in chapter 4, the number of nodes per task is not necessarily the same. It depends on the maximum degree allowed by degree scheduling algorithm and the degree of each nodes contained in the bloc. A task t contains consecutive nodes from task_tab[t].start to task_tab[t].end.

```
int main(int argc, char **argv) {
2 int nb_node = atoi(argv[2]), nb_edge,nb_th = atoi(argv[3]), chunk = atoi(argv[4]);
3 node_vect  = load_graph_txt(argv[1], &graph,nb_node,&nb_edge);
4 unsigned long long deg_sum = nb_edge*2;
5 task_tab  = generate_task_index(deg_sum, nb_node, nb_th, chunk, graph, node_vect);
6 Galois::for_each(graph_for_task.begin(), graph_for_task.end(), pref_attach_deg());
7 return 0;
8}
```

In the main part, degree-aware scheduling algorithm is called at line 5. At for_each loop, limits are changed from initial graph (in section 6.3.2 at line 4) to task graph (in this section at line 6). This change allows to have task granularity in loop body function described in the previous paragraph. There is graph loading instruction at line 3. Galois user has to decide how to implement graph ordering heuristics:

- During graph loading: in this case, degree-aware scheduling algorithm is associated to generate_task_index function.
- At graph processing level: in this case, degree-aware scheduling algorithm is separated to to generate_task_index function.

As explained in chapter 3, in order to have good performance, choosing one case or another depends on the target graph algorithm complexity. In this case with Preferential Attachment score, we do not performance issue (but illustration). So there is no problem to choose one or another.

Experimental Analysis with DSLs

In this section, we analyze experimental results got with each DSL (Galois and Green-Marl). For this analysis, as in the previous chapter, we present results got with Pagerank [35]. But in this case, we used implementations provided respectively by Galois and Green-Marl software. For Green-marl case, we consider new openMP (the one with taskloop). This implementations uses Yale representation (described in chapter 2) for graphs. Experiments were made on Numa4 machine also described in chapter 2 . Linux perf is used to report cache misses and cache references. Live Journal and Orkut datasets [54] are also used. As a reminder, Live Journal is an unoriented graph with 3,997,962 nodes and 34,681,189(X2) edges (its size is 279.84 MB); Orkut is an unoriented graph with 3,072,441 nodes and 117,185,083(X2) edges (its size is 905.77 MB).

In order to carry out this analysis, we will study one by one cache references reduction, cache misses reduction and then their impact in execution time reduction. We try to answer the following questions:

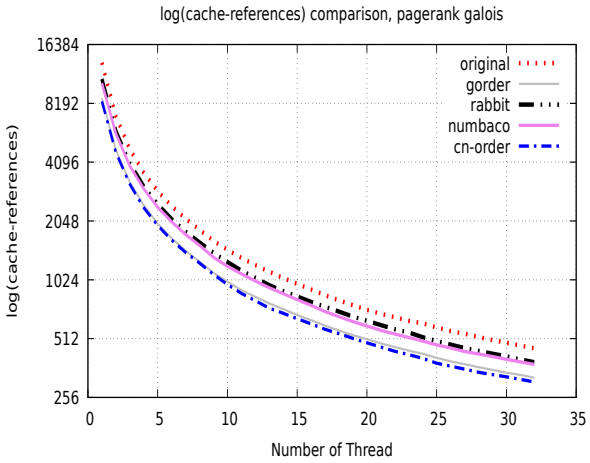


Figure 6.4: cache-ref with graph orders – Orkut

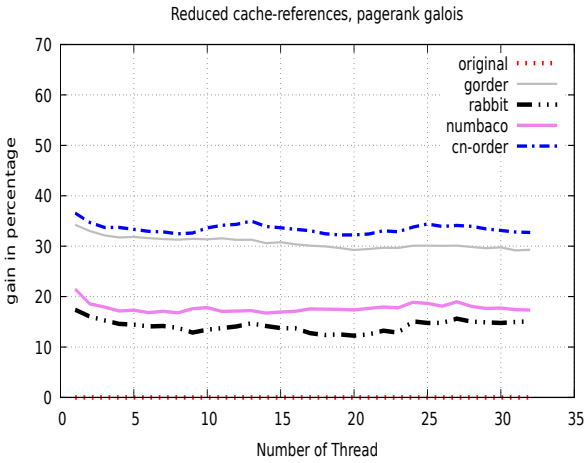


Figure 6.5: cache-ref with graph orders – Orkut

- Which DSL gives the best results?
- In terms of performance, do both DSLs have the same behavior with graph ordering heuristics and degree-aware scheduling heuristics?
- What is the difference with non-DSLs results?

6.4 Cache References Reduction in DSLs

Section 6.4.1 and section 6.4.2 present in details cache references reduction respectively in Galois and in Green-Marl.

6.4.1 Cache References Reduction in Galois

Figures 6.4, 6.5, 6.6, 6.7 (reported with Orkut dataset) and figures 6.8, 6.9, 6.10, 6.11 (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).

Orkut Dataset. With graph ordering heuristics in figures 6.4 and 6.5, Cn-order is the best with 36% to 33% of reduced cache references. It is close to Goder with 33% to 29%. NumBaCo is the third with around 21% to 19% and Rabbit is the last with 18% to 13%. The gained percentage is nearly the same from one thread to 32 threads.

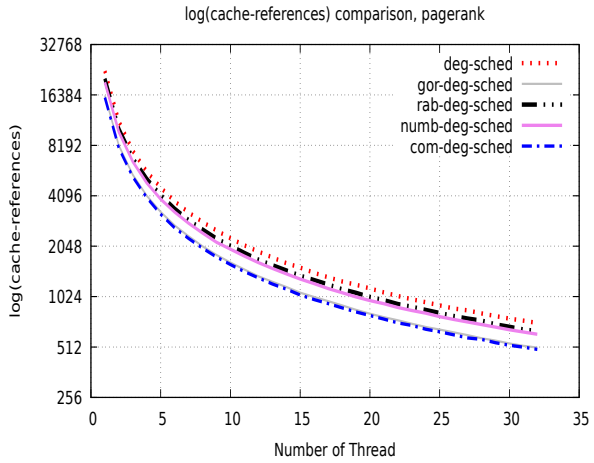


Figure 6.6: cache-ref with scheduling – Orkut

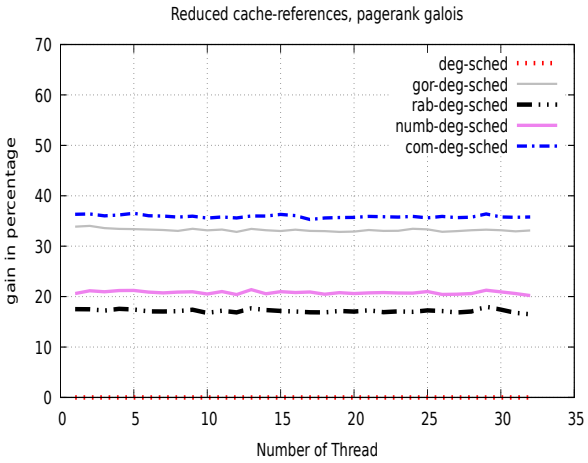


Figure 6.7: cache-ref with scheduling – Orkut

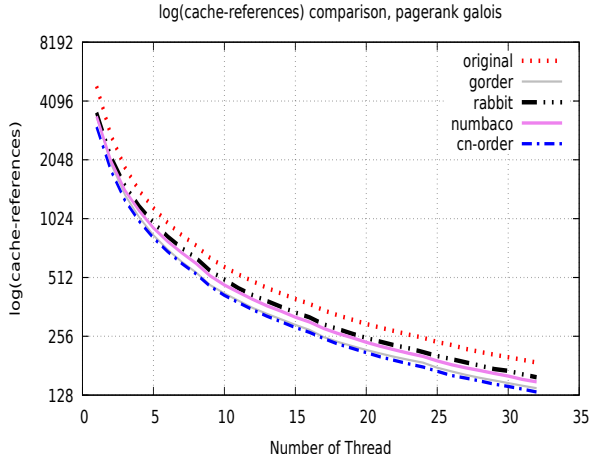


Figure 6.8: cache-ref with graph orders – Lj

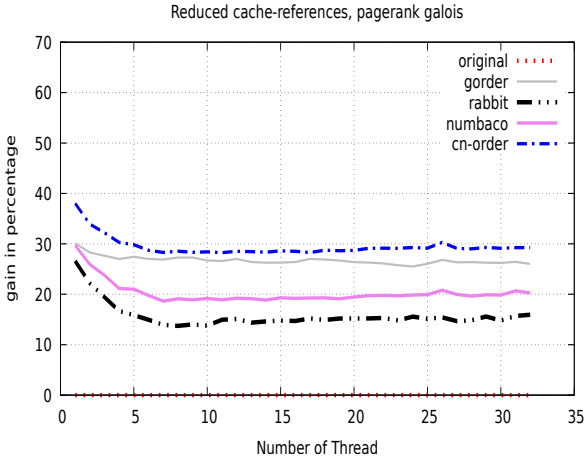


Figure 6.9: cache-ref with graph orders – Lj

With degree-aware scheduling heuristics in figures 6.6 and 6.7, Comm-deg-scheduling is the best with almost 36% of reduced cache references. Gor-deg-scheduling is the second with 33%, Numb-deg-scheduling is the third with almost 21% and Rab-deg-scheduling is the last with 17%.

Live Journal Dataset. In figures 6.8 and 6.9, with graph ordering heuristics, Cn-order reduces cache references between 29% and 38%. Goder varies between 27% and 30 %. NumBaCo varies between 19% and 29%. Rabbit varies between 16% and 26%.

With degree-aware scheduling heuristics in figures 6.10 and 6.11, Comm-deg-scheduling reduces cache references between 35% and 36%. Gor-deg-scheduling varies between 28%

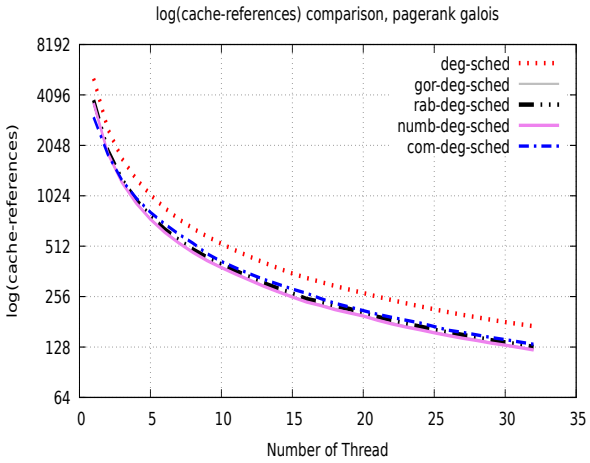


Figure 6.10: cache-ref with scheduling – Lj

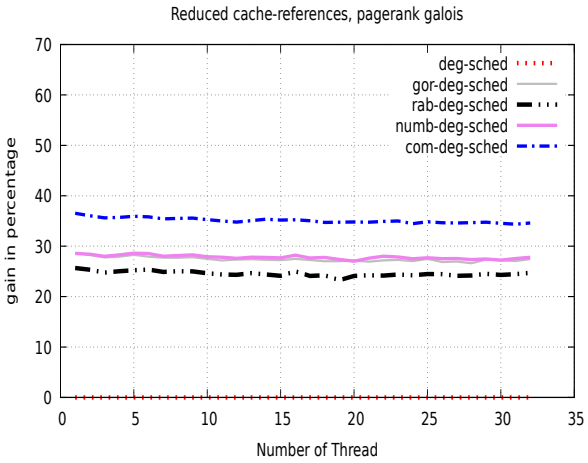


Figure 6.11: cache-ref with scheduling – Lj

and 29%. Numb-deg-scheduling varies between 28% and 29%. Rab-deg-scheduling varies between 25% and 26%.

6.4.2 Cache References Reduction in Green-Marl

Figures 6.12, 6.13, 6.14, 6.15 (reported with Orkut dataset) and figures 6.16, 6.17, 6.18, 6.19 (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).

Orkut Dataset. With graph ordering heuristics in figures 6.12 and 6.13, Cn-order is the best with almost 43% of reduced cache references. Gorder is second with almost 41%. NumBaCo is the third with almost 22% and Rabbit is the last with almost 18%.

With degree-aware scheduling heuristics in figures 6.14 and 6.15, performances are very close to what we got with graph ordering heuristics: Comm-deg-scheduling is the best with 43% of reduced cache references; Gor-deg-scheduling is the second with 42%, Numb-deg-scheduling is the third with almost 22% and Rab-deg-scheduling is the last with 18%.

Live Journal Dataset. In figures 6.16 and 6.17, with graph ordering heuristics, Cn-order reduces cache references by almost 40%. Goder varies between 26% and 27%. NumBaCo varies between 27% and 29%. Rabbit varies between 19% and 21%.

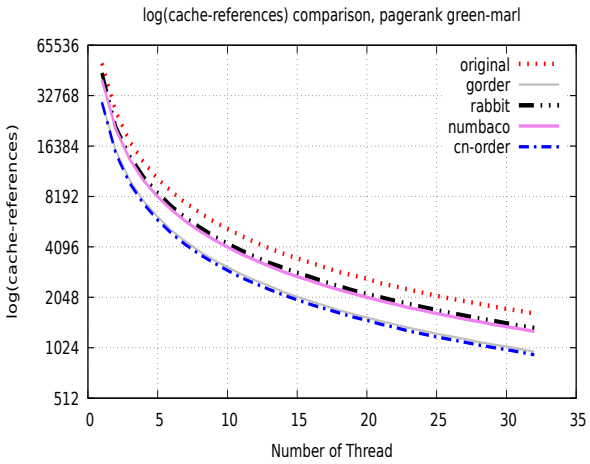


Figure 6.12: cache-ref with graph orders – Orkut

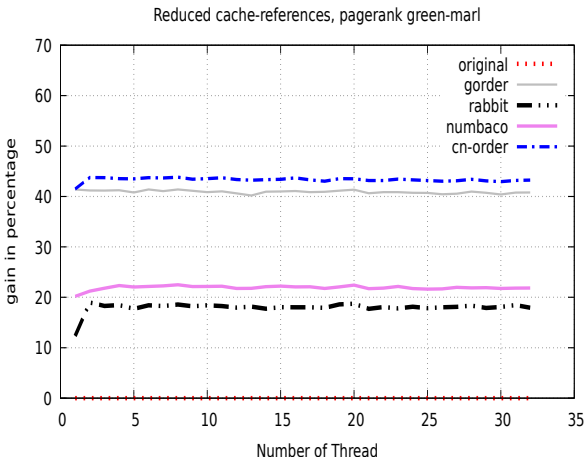


Figure 6.13: cache-ref with graph orders – Orkut

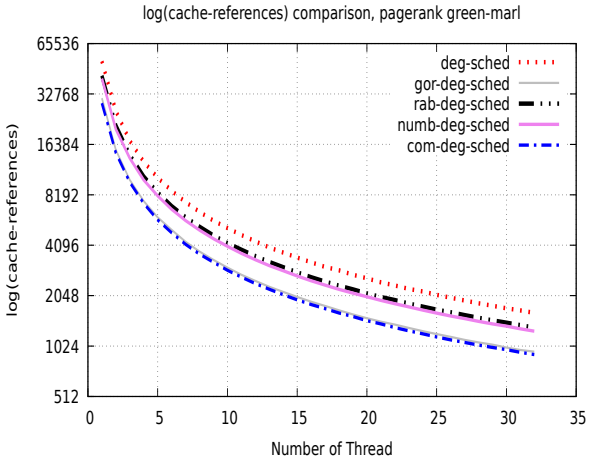


Figure 6.14: cache-ref with scheduling – Orkut

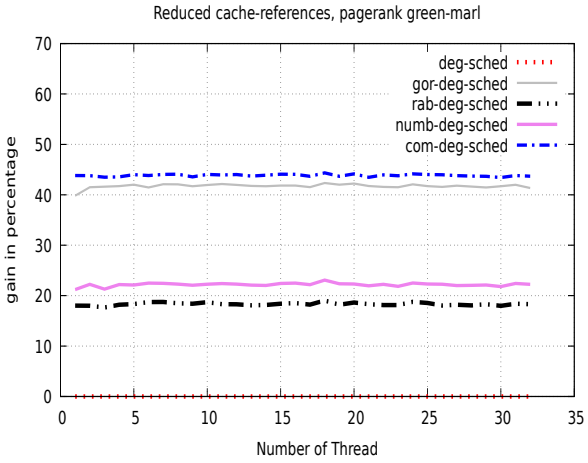


Figure 6.15: cache-ref with scheduling – Orkut

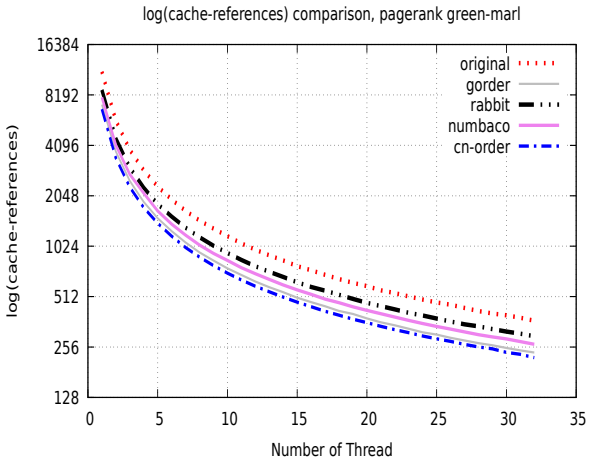


Figure 6.16: cache-ref with graph orders – Lj

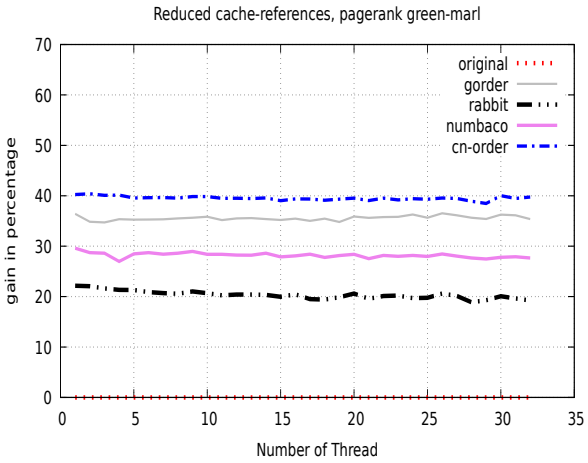


Figure 6.17: cache-ref with graph orders – Lj

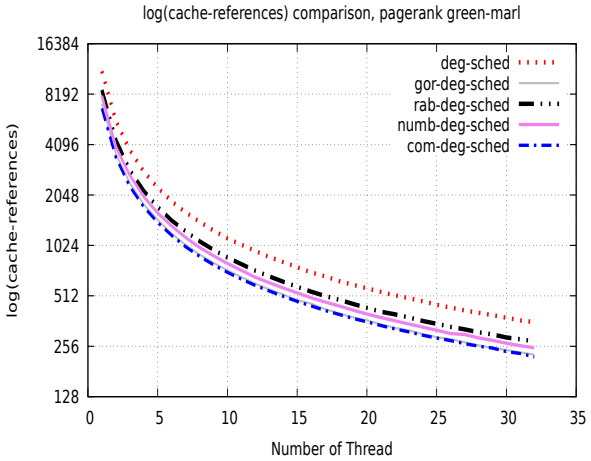


Figure 6.18: cache-ref with scheduling – Lj

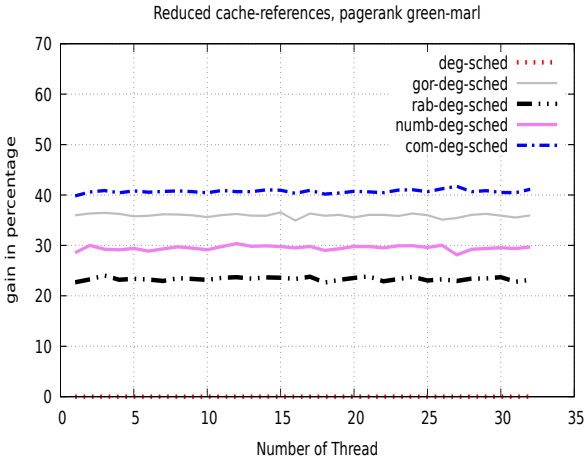


Figure 6.19: cache-ref with scheduling – Lj

As with Orkut dataset, with degree-aware scheduling heuristics in figures 6.18 and 6.19, performances are close to what we got with graph ordering heuristics: Comm-deg-scheduling reduces cache references by almost 40%. Gor-deg-scheduling reduces by 26%. Numb-deg-scheduling varies between 29% and 30%. Rab-deg-scheduling varies between 22% and 23%.

Section Summarize. In this section, we showed the following points:

- We have the same behavior of heuristics performance (in terms of cache references reduction) in both Galois and Green-Marl. This means Green-Marl and Galois does not have a big influence in cache references reduction induced by graph ordering heuristics and degree-aware scheduling heuristics. In other words, we showed in

this section that:

  - With graph ordering heuristics implemented in Galois and Green-Marl, Cn-order is the best, Gorder is the second, NumBaCo is the third and Rabbit is the last.
  - Degree-aware scheduling heuristics implemented in Green-Marl and Galois follow their homologous graph ordering heuristics: Comm-deg-scheduling is the best, Gor-deg-scheduling is the second, Numb-deg-scheduling is the third and Rab-deg-scheduling is the last.

- This results (summarized in table 6.1 for Cn-order and Comm-deg-scheduling) showed that Galois and Green-Marl produce better performance in terms of cache references reduction compared to results we got in chapter 4. And when comparing Galois to Green-Marl, we see that Green-Marl produces better results.

Table 6.1: Cache references reduction in Galois and Green-marl

DSL	No DSL		Galois		**Green-Marl**	
Heuristic	Cn-order	Com-deg	Cn-order	Com-deg	Cn-order	Com-deg
% (Orkut)	30%	30%	36% - 33%	36%	**43%**	**43%**
% (Lj)	32% - 35%	30% - 35%	29% - 38%	35% - 36%	**40%**	**40%**

6.5 Cache Misses Reduction in DSLs.

This section present cache misses reduction with Galois and Green-Marl.

6.5.1 Cache Misses Reduction in Galois

Cache misses reported with Orkut dataset are presented in figures 6.20, 6.21, 6.22, 6.23 and with Live Journal dataset in figures 6.24, 6.25, 6.26, 6.27. As with cache references reduction section, 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).

Orkut Dataset. In figures 6.20 and 6.21, Cn-order has the best cache misses reduction with 40% to 52%. The other heuristics are switching their position according to the number of threads. With NumBaCo, cache misses are reduced from 29% to 49%. With

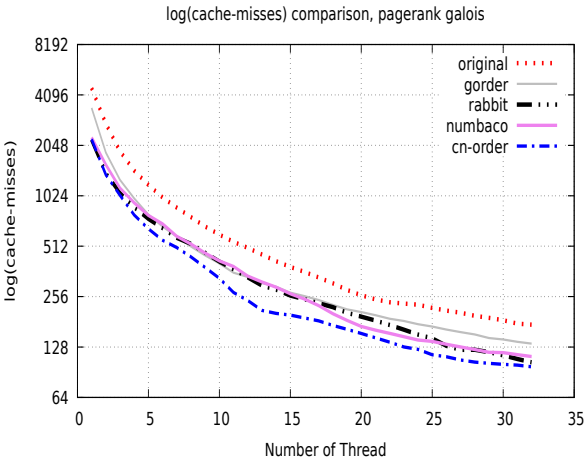


Figure 6.20: cache-misses with graph orders – Orkut

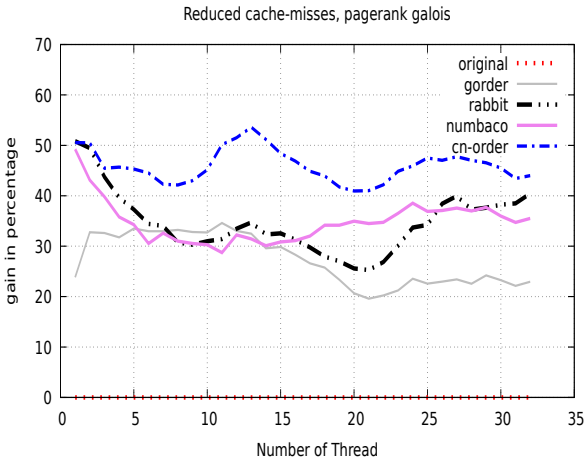


Figure 6.21: cache-misses with graph orders – Orkut

Rabbit, we have 26% to 50% and with Gorder we have 20% to 34% of cache misses reduction.

In figures 6.22 and 6.23, Comm-deg-scheduling reduces cache misses from 39% to 52%. Numb-deg-scheduling reduces cache misses from 28% to 49%, Rab-deg-scheduling from 29% to 50% and Gor-deg-scheduling from 23% to 31%.

Live Journal Dataset. In figures 6.24 and 6.25, Cn-order has the best cache misses reduction with almost 37% to 50%. NumBaCo is the second with 36% to 50%. Rabbit is the third with 33% to 44%. Gorder is the last with 19% to 21%.

In figures 6.26 and 6.27, Comm-deg-scheduling and Numb-deg-scheduling are the best with 35% - 45% of reduced cache misses. Rab-deg-scheduling (with 31% – 41%) is better than Gor-deg-scheduling (with 17% to 22%).

6.5.2 Cache Misses Reduction in Green-Marl

Cache misses reported with Orkut dataset are presented in figures 6.28, 6.29, 6.30, 6.31 and with Live Journal dataset in figures 6.32, 6.33, 6.34, 6.35. As with cache references reduction section, 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).

Orkut Dataset. In figures 6.28 and 6.29, Cn-order has the best cache misses reduction with 52%. NumBaCo is the second with 50% of cache misses reduction. Rabbit is the

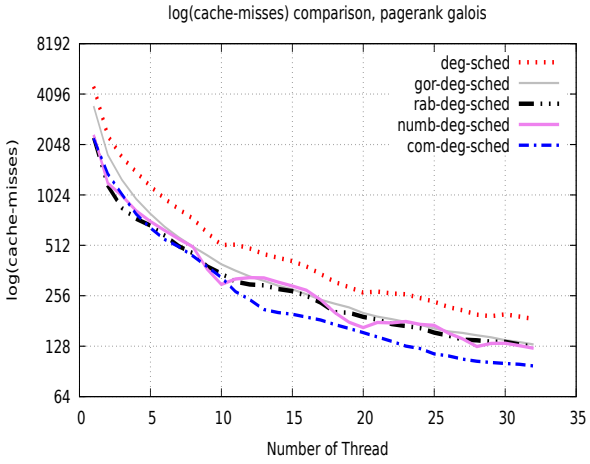


Figure 6.22: cache-misses with scheduling – Orkut

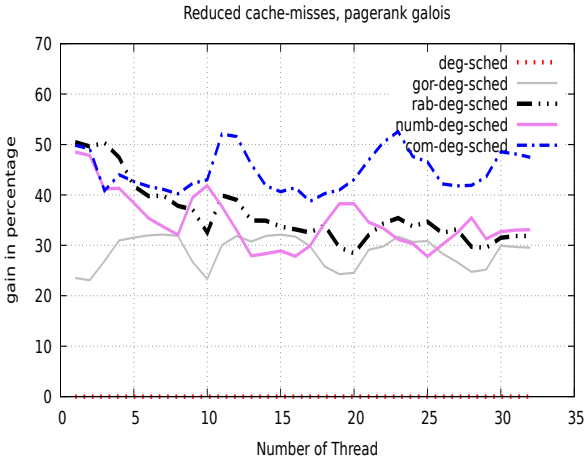


Figure 6.23: cache-misses with scheduling – Orkut

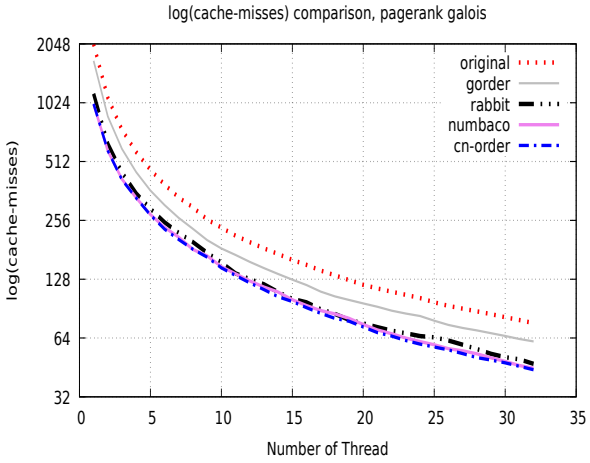


Figure 6.24: cache-misses with graph orders – Lj

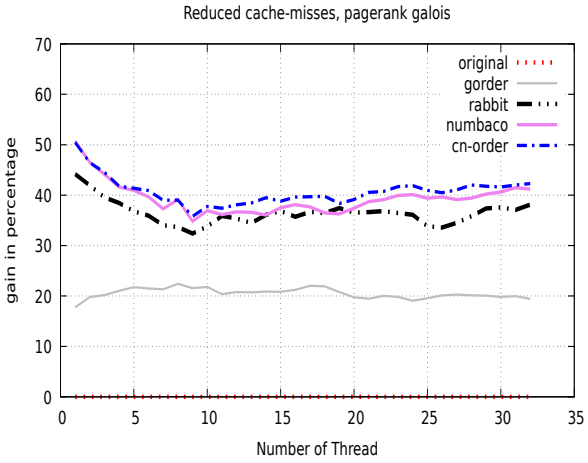


Figure 6.25: cache-misses with graph orders – Lj

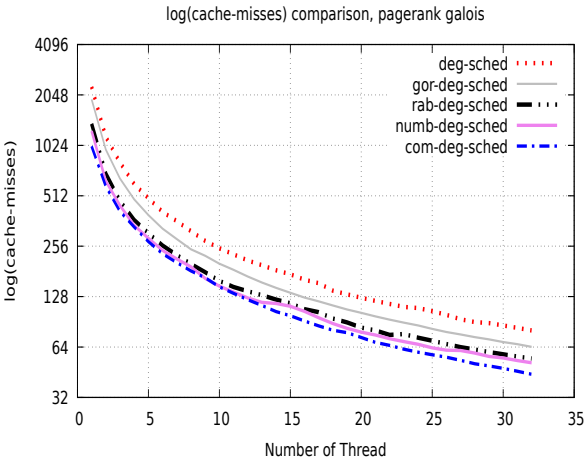


Figure 6.26: cache-misses with scheduling – Lj

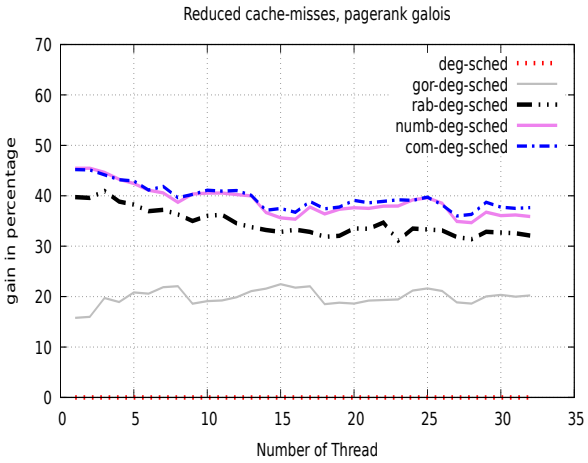


Figure 6.27: cache-misses with scheduling – Lj

third with 47% and Gorder is the last with 9% to 19% of reduced cache misses.

In figures 6.30 and 6.31, degree-aware scheduling heuristics have the same behavior as their homologous of graph ordering heuristics. Comm-deg-scheduling is the best with 52% of reduced cache misses. Numb-deg-scheduling is the second with 50%, Rab-deg-scheduling is the third with 47% and Gor-deg-scheduling is the last with 9% to 16% of reduced cache misses.

Live Journal Dataset. In figures 6.32 and 6.33, Cn-order and NumBaCo have the best cache misses reduction with almost 38% to 46%. Rabbit (with almost 24% to 32%) is better than Gorder (with 17% to 18%).

In figures 6.34 and 6.35, as for Orkut dataset, degree-aware scheduling heuristics (implemented in Green-Marl) have also the same behavior as their homologous of graph ordering heuristics in cache misses reduction. Comm-deg-scheduling and Numb-deg-scheduling are the best with 40% to 46% of reduced cache misses. Rab-deg-scheduling (with 25% to 32%) is better than Gor-deg-scheduling (with 16% to 19%).

Section Summarize. In this section, we showed the following points:

- Like in terms of cache references reduction, both Galois and Green-Marl do not have a great influence in cache misses reduction induced by graph ordering heuristics and degree-aware scheduling heuristics. As what we got without DSL in chapter 4, we showed in this section that:

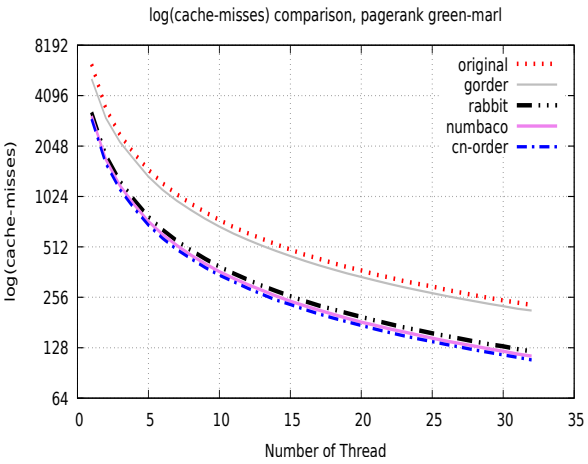


Figure 6.28: cache-misses with graph orders – Orkut

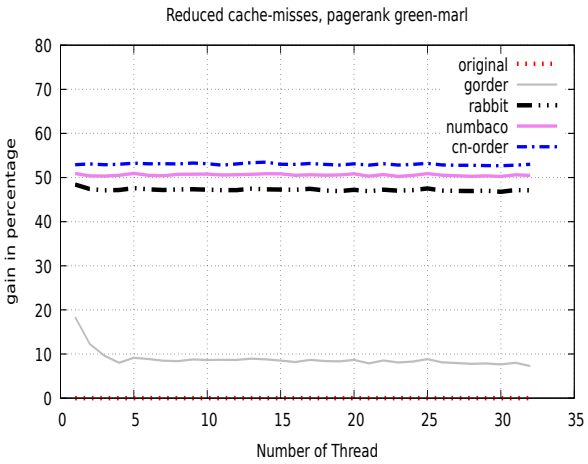


Figure 6.29: cache-misses with graph orders – Orkut

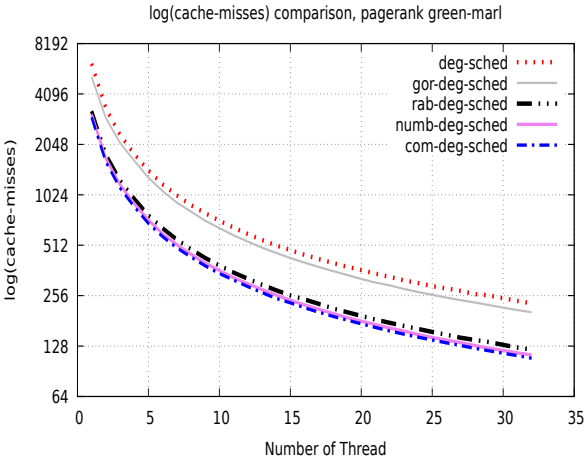


Figure 6.30: cache-misses with scheduling – Orkut

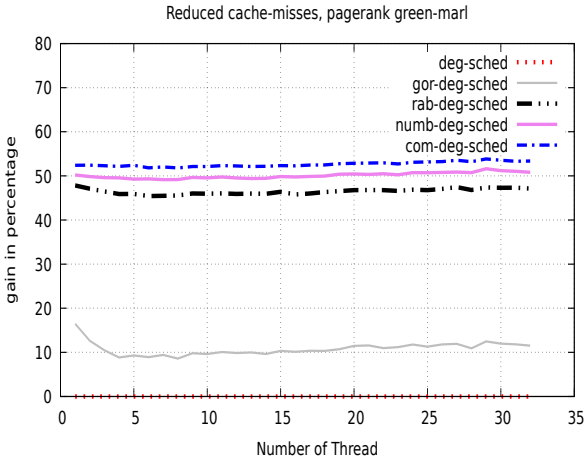


Figure 6.31: cache-misses with scheduling – Orkut

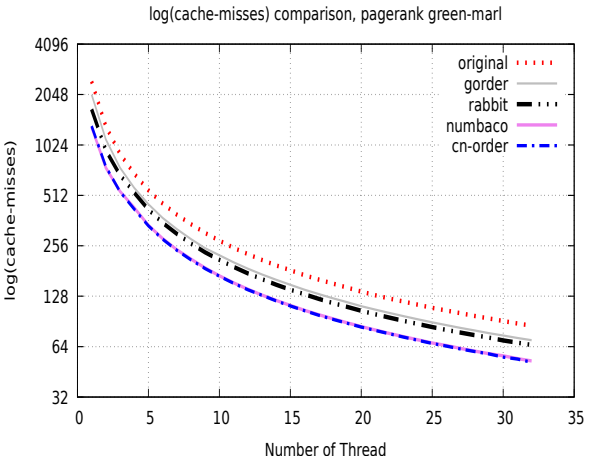


Figure 6.32: cache-misses with graph orders – Lj

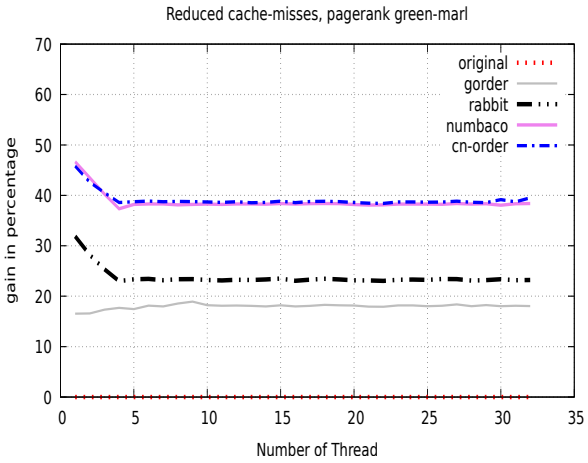


Figure 6.33: cache-misses with graph orders – Lj

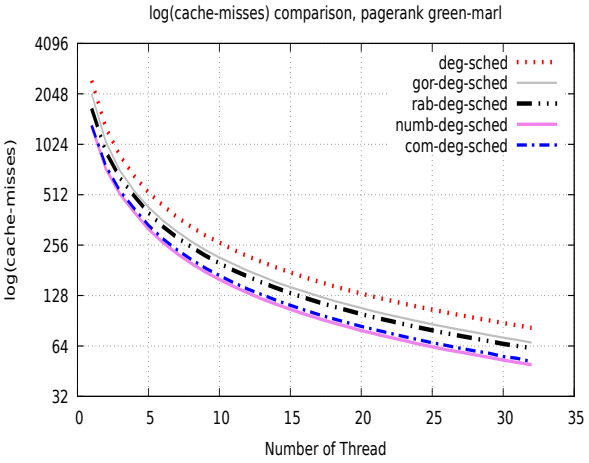


Figure 6.34: cache-misses with scheduling – Lj

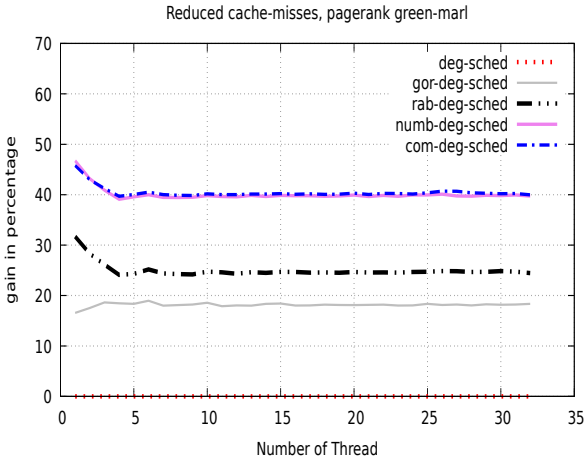


Figure 6.35: cache-misses with scheduling – Lj

  - With graph ordering heuristics implemented in Galois and Green-Marl, generaly, Cn-order is the best, NumBaCo is the second, Rabbit is the third and Gorder is the last.
  - Degree-aware scheduling heuristics implemented in Green-Marl and Galois follow their homologous graph ordering heuristics: Comm-deg-scheduling is the best, Numb-deg-scheduling is the second, Rab-deg-scheduling is the third and Gor-deg-scheduling is the last.

- Results (summarized in table 6.2 for Cn-order and Comm-deg-scheduling) showed that, in general case, Galois and Green-Marl produce better performance in terms of cache misses reduction compared to results we got in chapter 4. And when comparing Galois to Green-Marl, we see that Green-Marl usualy produces better results.

Table 6.2: Cache misses reduction in Galois and Green-marl

DSL	No DSL		Galois		**Green-Marl**	
Heuristic	Cn-order	Com-deg	Cn-order	Com-deg	Cn-order	Com-deg
% (Orkut)	38% - 45%	37% - 45%	40% - 52%	39% - 52%	**52%**	**52%**
% (Lj)	**38%** - 44%	30% - 45%	37% - **50%**	35% - 45%	**38%** - 46%	**40% - 46%**

Which impact do cache misses reduction and cache references reduction have compared to what we got in chapter 4? The following section tries to answer this question.

6.6 Impact in time reducing

Section 6.6.1 presents time reduced with Green-Marl DSL and section 6.6.2 presents time reduced with Galois.

6.6.1 Impact in time reducing with Green-Marl

Figures 6.36, 6.37, 6.38, 6.39 (gotten with Live Journal dataset) and figures 6.40, 6.41, 6.42, 6.43 (gotten with Orkut dataset) represent time reduction due to graph ordering heuristics (figures 6.36, 6.37, 6.40, 6.41) and degree-aware scheduling heuristics (figures 6.38, 6.39, 6.42, 6.43).

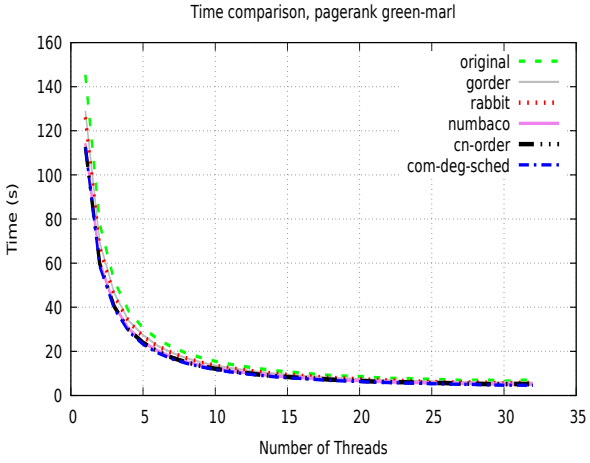


Figure 6.36: time with graph orders – Lj

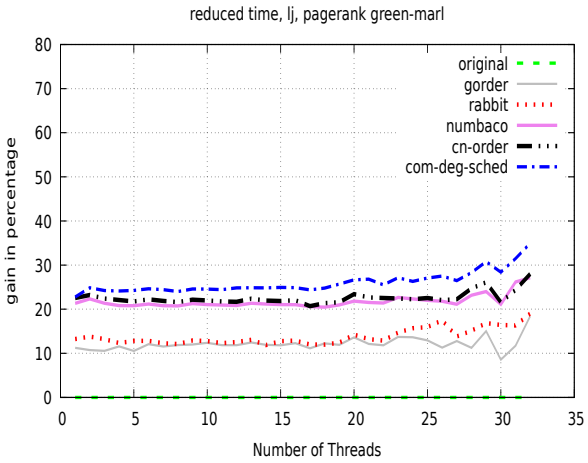


Figure 6.37: time reduced with graph orders – Lj

Live Journal Dataset. In figures 6.36 and 6.37, when comparing graph ordering heuristics, Cn-order is the best with 22% to 28% in time reduction; NumBaCo is the second with 21% to 27% (close to Cn-order). Rabbit is the third with 11% to 19% and Gorder is the last with 9% to 18% in time reduction.

Impact of degree-aware scheduling is visible with Comm-deg-sched that produces the best results when it is compared to graph ordering heuristics (23% to 35%).

In figures 6.38 and 6.39, we compare degree-aware scheduling heuristics. If Comm-deg-sched curve remains almost always the best, it is very close to Numb-deg-sched curve (23% to 35%). This observation suggests that both heuristic categories ensure performances.

Orkut Dataset. In figures 6.40 and 6.41, we have the same behavior as with Live Journal dataset: Cn-order is the best with 27% to 35%, NumBaCo is the second with 24% to 31%, Rabbit is the third with 16% to 26% and Gorder is the last with 16% to 21% in time reduction. The difference with the previous is that here, Comm-deg-sched have the same performance with Cn-order.

In figures 6.42 and 6.43, Comm-deg-sched produces the best results with 27% to 35%, Numb-deg-sched is the second with 23% to 31%, Rab-deg-sched is the third with 20% to 27% and Gor-deg-sched is the last with 15% to 21%.

This section showed that the impact of cache misses reduction and cache references reduction is effective in Green-Marl. Both categories of heuristics allow to increase performances. The next section studies the impact of these heuristics with Galois.

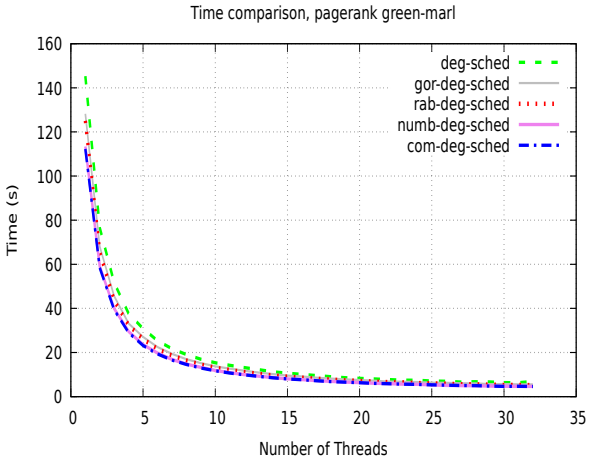


Figure 6.38: time with scheduling – Lj

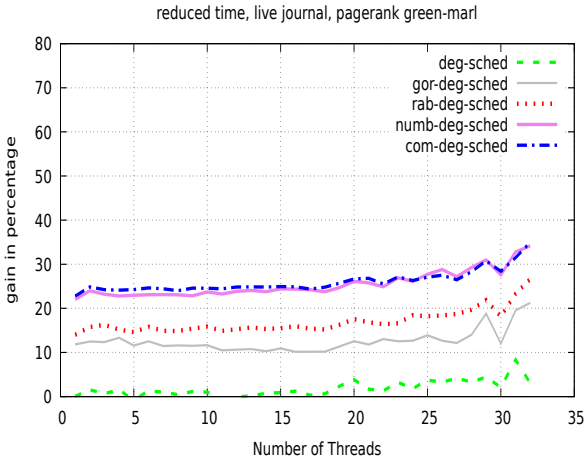


Figure 6.39: time reduced with scheduling – Lj

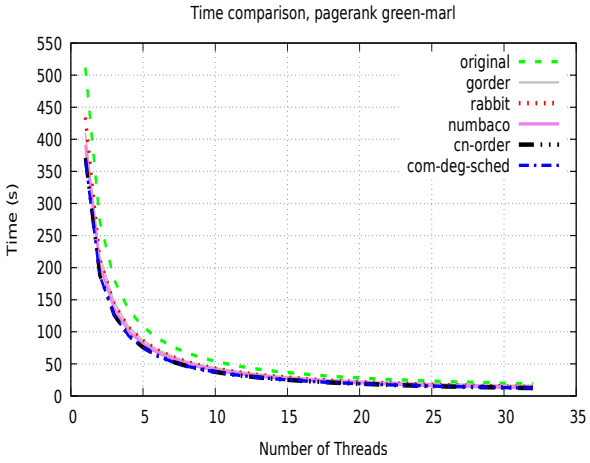


Figure 6.40: time with graph orders – Orkut

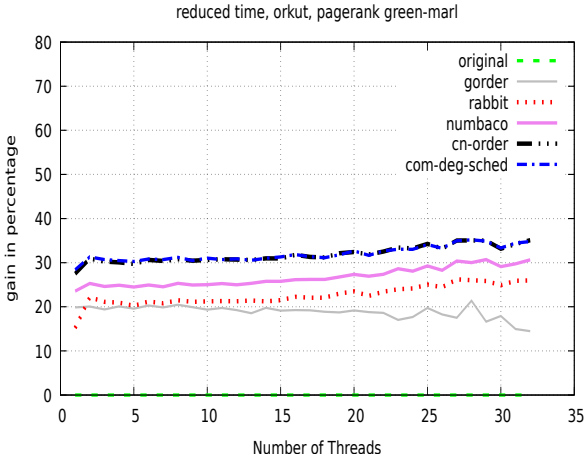


Figure 6.41: time reduced with graph orders – Orkut

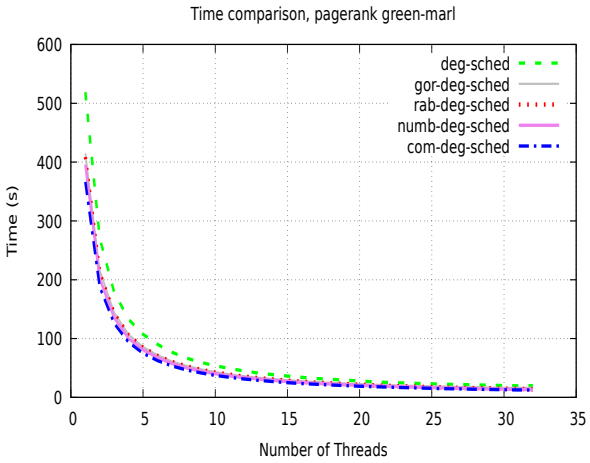


Figure 6.42: time with scheduling – Orkut

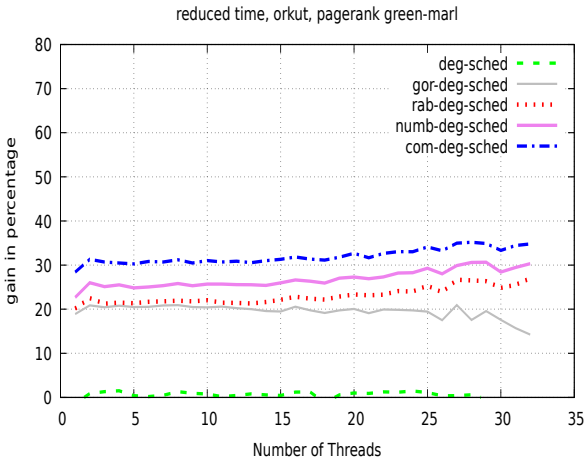


Figure 6.43: time reduced with scheduling – Orkut

6.6.2 Impact in time reducing with Galois

Figures 6.44, 6.45, 6.46, 6.47 (gotten with Live Journal dataset) and figures 6.48, 6.49, 6.50, 6.51 (gotten with Orkut dataset) represent time reduction due to graph ordering heuristics (figures 6.44, 6.45, 6.48, 6.49) and degree-aware scheduling heuristics (figures 6.46, 6.47, 6.50, 6.51).

Live Journal Dataset. In figures 6.44 and 6.45, we can see that, observed results are different to what we got with Green-Marl. Graph ordering heuristic curves are switching their position according to the number of threads. Here we present observations with one and 32 threads.

- Cn-order starts with 26% (one thread) and ends with 48% (32 threads).
- NumBaCo starts with 24% (one thread) and ends with 50% (32 threads).
- Rabbit starts with 21% (one thread) and ends with 48% (32 threads).
- Gorder starts with 10% (one thread) and ends also with 10% (32 threads).

Between 2 and 31 threads there many variations. As explained in chapter 4, these variations are due to imbalanced load among threads. While in that chapter using degree-aware scheduling heuristics fixes this problem and produces better results, with Galois, it doesn't and tends to produce worst results (look at curve Comm-deg-sched at figure 6.45).

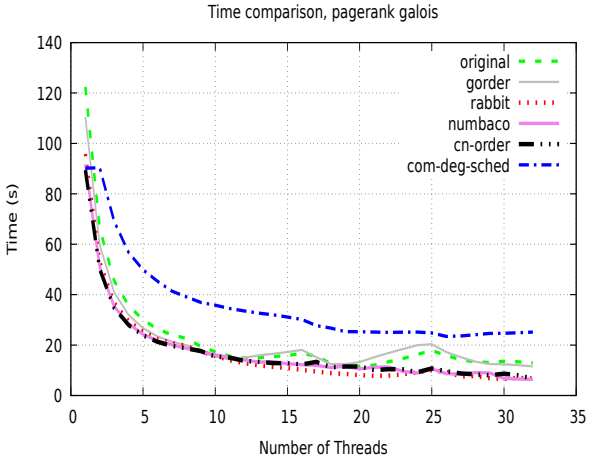


Figure 6.44: time with graph orders – Lj

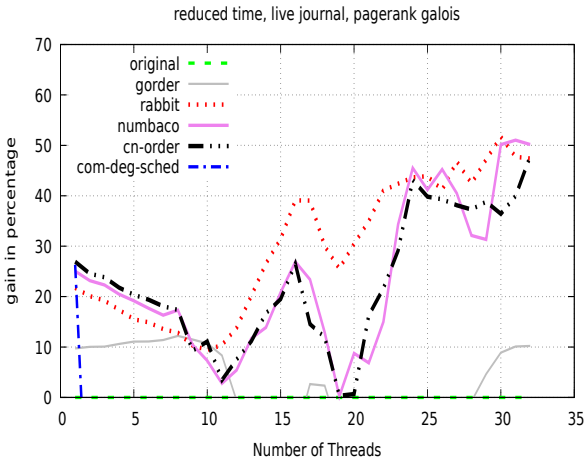


Figure 6.45: time reduced with graph orders – Lj

In figures 6.46 and 6.47, we can see that we got positive gain only with one thread. That means, our degree-aware scheduling heuristics are inefficient with Galois. The reason of this inefficiency is that Galois already has a scheduler that is in charge to ensure load balancing among threads. Results suggests that Galois scheduler is incompatible with our degree-aware scheduling heuristic.

The switching position observed with graph ordering heuristics according to the number of threads also suggests that Galois scheduler alone (that is without degree-aware scheduling heuristics) does not fix the imbalance load problem. One should see how to combine Galois scheduler and our scheduling heuristics in order to increase performances.

Orkut Dataset. In figures 6.48 and 6.49, observations are as follows.

- Cn-order starts with 27% (one thread) and ends with 30% (32 threads). It has the highest gain with 28 threads (38%).
- NumBaCo starts with 21% (one thread) and ends with 48% (32 threads).
- Rabbit starts with 19% (one thread) and ends with 46% (32 threads).
- Gorder starts with 16% (one thread) and ends with negative gain (32 threads). It has the highest gain with 8 threads (22%).

In figures 6.50 and 6.51, we have positive gain only with one thread. This confirms the results we got in Live Journal dataset. Galois needs only graph ordering heuristics

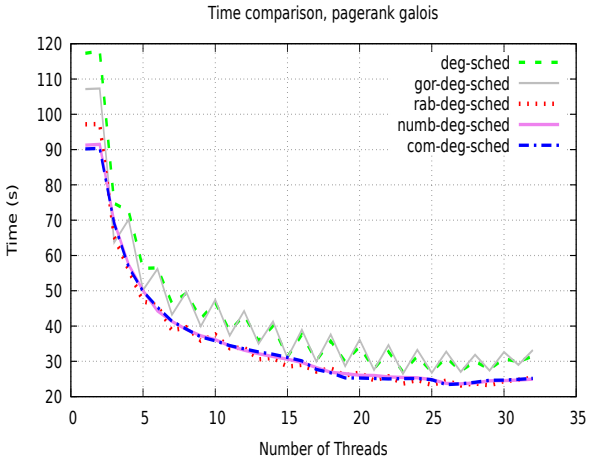


Figure 6.46: time with scheduling – Lj

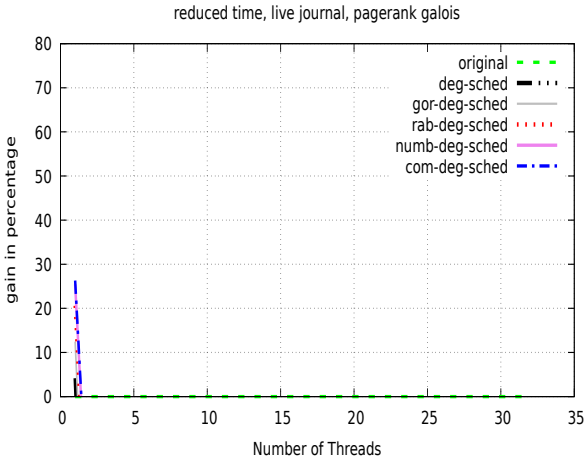


Figure 6.47: time reduced with scheduling – Lj

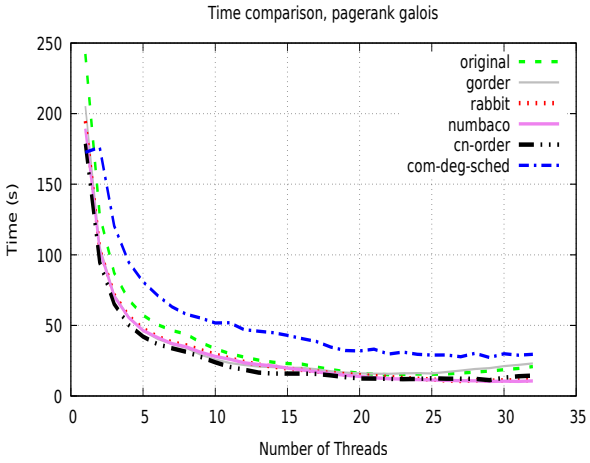


Figure 6.48: time with graph orders – Orkut

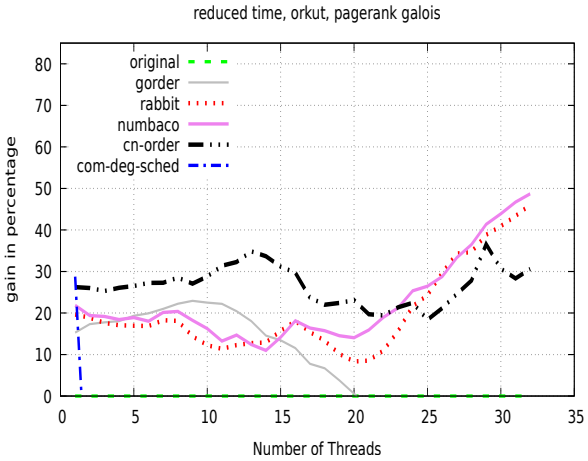


Figure 6.49: time reduced with graph orders – Orkut

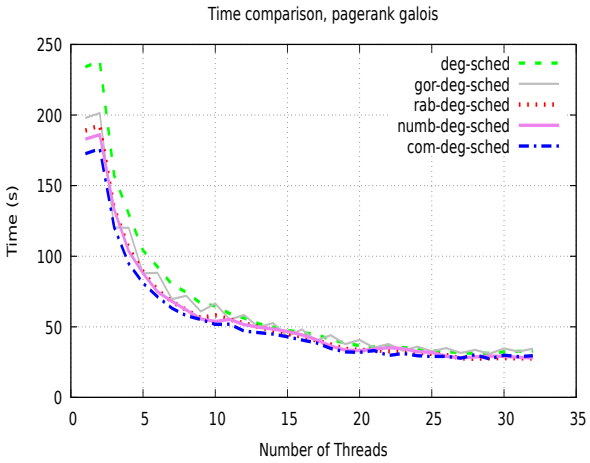


Figure 6.50: time with scheduling – Orkut

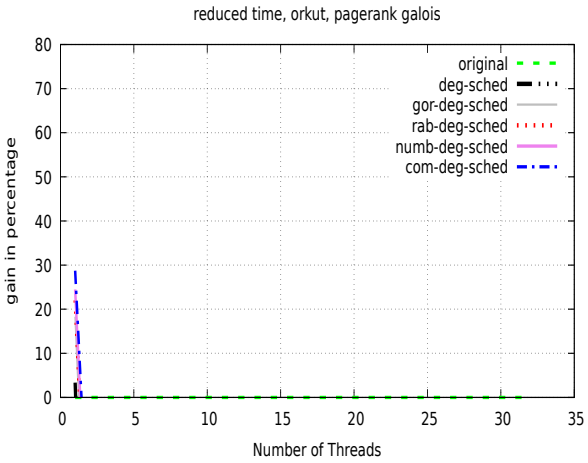


Figure 6.51: time reduced with scheduling – Orkut

to increase performances. In other words, Galois scheduler, as it is now designed, is incompatible with our scheduling heuristics. If someone wants to take advantage to our heuristics, he should modified Galois scheduler.

Section Summarize. In this section we saw the impact of proposed heuristics through cache misses reduction and cache references reduction in time reduction with Galois and Green-Marl. If Green-Marl performances are increased by both graph ordering heuristics and degree-aware scheduling heuristics, it is not the case with Galois. This last earns performances only with graph ordering heuristics. With degree-aware scheduling heuristics implemented in Galois, performances are even worst.

Table 6.3 presents time reduced with Galois, Green-Marl compared to results without DSL. This table revels these informations:

- Even if impact of cache misses and cache references reduction is higher without DSL, our heuristics implemented in DSLs allow to reduce execution time by 35% (Green-Marl) and 48% (Galois).
- There is no gain with degree-aware implemented in Galois. This is because Galois already has a scheduler. To take advantage to degree-aware scheduler, one should modify the actual Galois scheduler.

The next section will conclude the hole chapter and will give perspectives that can follow directly this work.

DSL	No DSL		Galois		**Green-Marl**	
Heuristic	Cn-order	Com-deg	Cn-order	Com-deg	Cn-order	Com-deg
% (Orkut)	no gain	**43%**	30%	no gain	35%	35%
% (Lj)	38%	**51%**	48%	no gain	28%	35%

Table 6.3: Time reduced (with 32 threads) in Galois and Green-marl

6.7 Discussion

Chapter Assessment. In this chapter we present the implementation of graph ordering heuristics and degree-aware scheduling heuristics in Galois and Green-Marl. With Green-Marl, performances are increased by both graph ordering heuristics and degree-aware scheduling heuristics. With Galois, performances are increased only with graph ordering heuristics. This is because Galois already has a scheduler; and our own scheduler implement at the top of Galois produces conflicts that decrease performance.

Chapter Perspectives. To take advantage to degree-aware scheduler, one should modify the actual Galois scheduler. That is design a new algorithm that takes advantage of both initial scheduler of Galois and our degree-aware scheduler.

Another future work is to implement these heuristics in other graph analysis platforms (evoked in the previous chapter). That will probably allow to increase their performances.

Next Chapter. The next chapter will summarize the whole thesis, our contribution and envisaged perspectives.

Conclusion

Chapter 7

Conclusion

In this thesis, we took up implementation and performance challenges of graph applications through parallelism and complex networks knowledge. Our goal was to provide a domain specific language that ensures easy and efficient complex networks analysis. To achieve this goal, we followed two steps. In the first step, we exploited some complex networks properties in order to design graph ordering heuristics and degree-aware scheduling heuristics. In the second step, we implemented these heuristics in existing graph DSLs.

7.1 Graph Ordering Heuristics.

The first contribution of this thesis showed the exploitation of community structure in order to design community-aware graph ordering heuristics for cache misses reduction. After formalizing the Numbering Graph Problem (NGP) for cache misses reduction as linear arrangement problem (a well known NP-Complete problem), we proposed an heuristic called NumBaCo to solve that problem. Then, we compared Numbaco to Gorder and Rabbit (which appeared in the literature at the same period NumBaCo was proposed). This comparison allowed to design Cn-order, another heuristic that combines advantages of the three algorithms (Gorder, Rabbit and NumBaCo) to solve the NGP. Experimental results with one thread on Core2, Numa4 and Numa24 (with Pagerank and livejournal for example) showed that cache misses reduction and execution time reduction was ensured and also, that Cn-order uses well the advantages of the other orders and outperforms them.

7.2 Degree-aware Scheduling Heuristics.

The previous analysis stayed at only one thread. The second contribution this thesis considered the case of multiple threads applications. In that case, cache misses reduction is not sufficient to ensure execution time reduction; one should also take into account load balancing among threads. In that way, heterogeneity of node degree was used in order to design Deg-scheduling, a heuristic to solve degree-aware scheduling problem (formalized as the NP-Complete Knapsack problem). Deg-scheduling was combined to Cn-order, NumBaCo, Rabbit, and Gorder to form respectively Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling.

Experimental results with many threads on Numa4 (with Pagerank and livejournal for example) showed that Degree-aware scheduling heuristics (Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling) outperform their homologous graph ordering heuristics (Cn-order, NumBaCo, Rabbit, and Gorder) when they are compared two by two. This is because degree-aware scheduling heuristics ensure both cache misses reduction and load balancing among threads, while graph ordering heuristics ensure only cache misses reduction.

7.3 Heuristics integration in Graph DSLs.

The last contribution of this thesis the implementation of graph ordering heuristics and degree-aware scheduling heuristics parallel graph DSLS. After proposing a general methodology of heuristics integration in parallel graph DSLs, we implement the proposed heuristics into Green-Marl and Galois. Experimental results showed that with Green-Marl, performances are increased by both graph ordering heuristics and degree-aware scheduling heuristics (time was reduced by 35% due to heuristics). But with Galois, performances are increased only with graph ordering heuristics (time was reduced by 48% due to heuristics).

The results gotten in this thesis confirm that the performance of complex network analysis can be improved thanks to a proper parallelism combined to a proper exploitation of some complex network properties. This thesis also confirms that, instead of building a new DSL for complex network analysis, one can improve the existing graph DSLs. This improvement is done by integrating into these DSLs, heuristics that can make the applications implemented with them faster. However, the work presented in this thesis is still improvable. In the next section, we will present perspectives that follow directly this thesis.

7.4 Perspectives

General Perspectives. In this thesis we use two complex networks properties community structure and heterogeneity of node degree. Others properties like small word effect or transitivity can also be studied to design heuristics that improve graph applications performance. Instead of using complex networks properties to design heuristics, one can imagine to use machine learning.

Another perspective concerns the theoretical aspect of this thesis. Remember we showed that graph ordering for cache misses reduction and degree-aware scheduling for load balancing problems are NP-complete. We provided heuristics to solve them. But we didn't show how far these heuristics are to the optimal solutions. It is good to know it in the nearest future.

Perspectives in Graph Ordering Heuristics. The first contribution of this thesis focuses on graph ordering for cache misses reduction. In order to design new heuristics, one can compare theoretically and experimentally graph ordering for cache misses reduction with graph ordering for compression.

We also mentioned that Gorder can be improved by using item-set strategy (introduced by Agrawal *et al.* [1]). And even NumBaCo and Rabbit can also be improved:

- By using local communities detection to assign numbers inside the community.
- By using other communities detection algorithms (we used GCA and Louvain).

Perspectives in Heuristics Integration. We showed that with Galois DSL, we have no gain with scheduling heuristics. To take advantage to our scheduling heuristics, one should design a new algorithm that takes advantage of both internal scheduler of Galois and our degree-aware schedulers.

Another work to do in a near future is to implement these heuristics in other graph analysis platforms (evoked in chapter 5, section 5.1). That will probably contribute to increase their performances.

Bibliography

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[2] 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.

[3] 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.

[4] 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.

[5] Jay Banerjee, Won Kim, S-J Kim, and Jorge F. Garza. Clustering a dag for cad databases. *IEEE Transactions on Software Engineering*, 14(11):1684–1699, 1988.

[6] 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.

[7] 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.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, pages 587–596. ACM, 2011.

[9] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *ACM Sigplan Notices*, volume 45, pages 835–847. ACM, 2010.

[10] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228. ACM, 2009.

[11] Iain S Duff. Computer solution of large sparse positive definite systems (alan george and joseph w. liu. *SIAM Review*, 26(2):289–291, 1984.

[12] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

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

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

[15] Leonhard Euler. Solutio problematis ad geometriam situs pertinensis. *Comm. Acad. Sci. Imper. Petropol.*, 8:128–140, 1736.

[16] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

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

[18] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[19] 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.

[20] U Kang and Christos Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 300–309. IEEE, 2011.

[21] 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.

[22] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schneider, and Steven Völkel. Design guidelines for domain specific languages. In Matti Rossi, Jonathan Sprinkle, Jeff Gray, and Juha-Pekka Tolvanen, editors, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pages 7–13, 2009.

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

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

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

[26] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.

[27] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *journal of the Association for Information Science and Technology*, 58(7):1019–1031, 2007.

[28] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.

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

[30] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[31] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

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

[33] 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.

[34] Thomas Messi Nguélé. Langage dédié pour la fouille des réseaux sociaux. *Université de Yaoundé 1, Département d'Informatique*, Decembre 2013. Mémoire de master 2 recherche.

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

[36] Jordi Petit. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics (JEA)*, 8:2–3, 2003.

[37] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM Sigplan Notices*, 46(6):12–25, 2011.

[38] 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.

[39] 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.

[40] Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *Journal of Experimental Algorithmics (JEA)*, 13:4, 2009.

[41] Ilya Safro and Boris Temkin. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms*, 9(2):190–202, 2011.

[42] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

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

[44] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[45] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[46] Jyothish Soman and Ankur Narang. Fast community detection algorithm with gpus and multicore architectures. *IEEE International Parallel Distributed Processing Symposium*, 2011.

[47] 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.

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

[49] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[50] Paolo Suppa and Eugenio Zimeo. A clustered approach for fast computation of betweenness centrality in social networks. In *Big Data (BigData Congress), 2015 IEEE International Congress on*, pages 47–54. IEEE, 2015.

[51] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[52] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[53] 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.

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

Appendix A

Examples With Karate Graph

A.1 Karate Hierarchy With Louvain

Figure A.2 show karate hierarchy.

A.2 Karate With Different Orders

A.2.1 Karate With Goder

With gorder in figure A.3, nodes belonging to the same community in figure A.1 do not necessarily have consecutive numbers. For example, node 1 in community **C4** is followed by node 2 which is community **C2**.

A.2.2 Karate With Rabbit Order

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

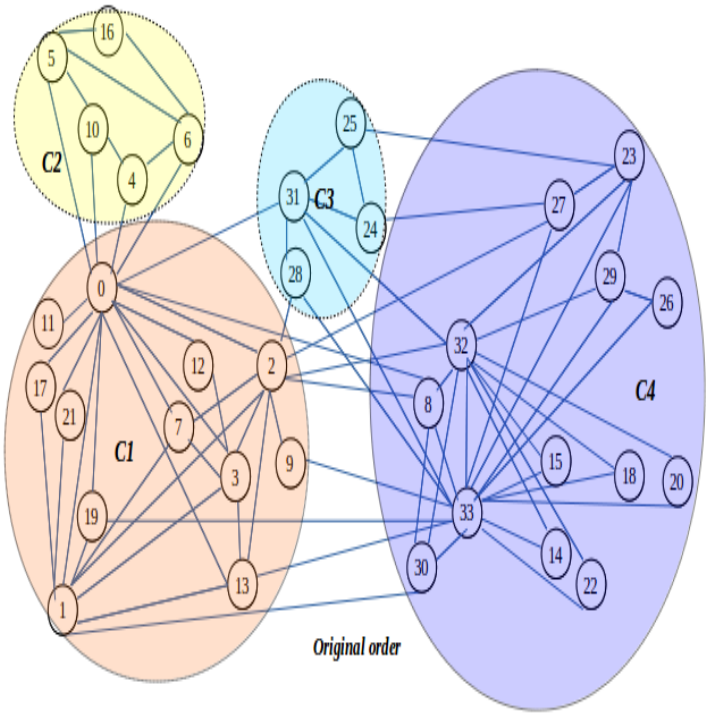


Figure A.1: karate graph

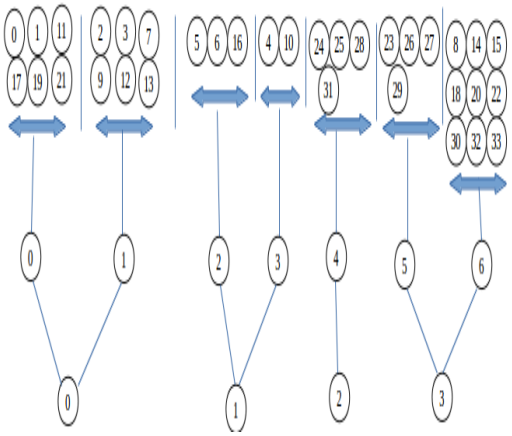


Figure A.2: Hierarchy with Louvain

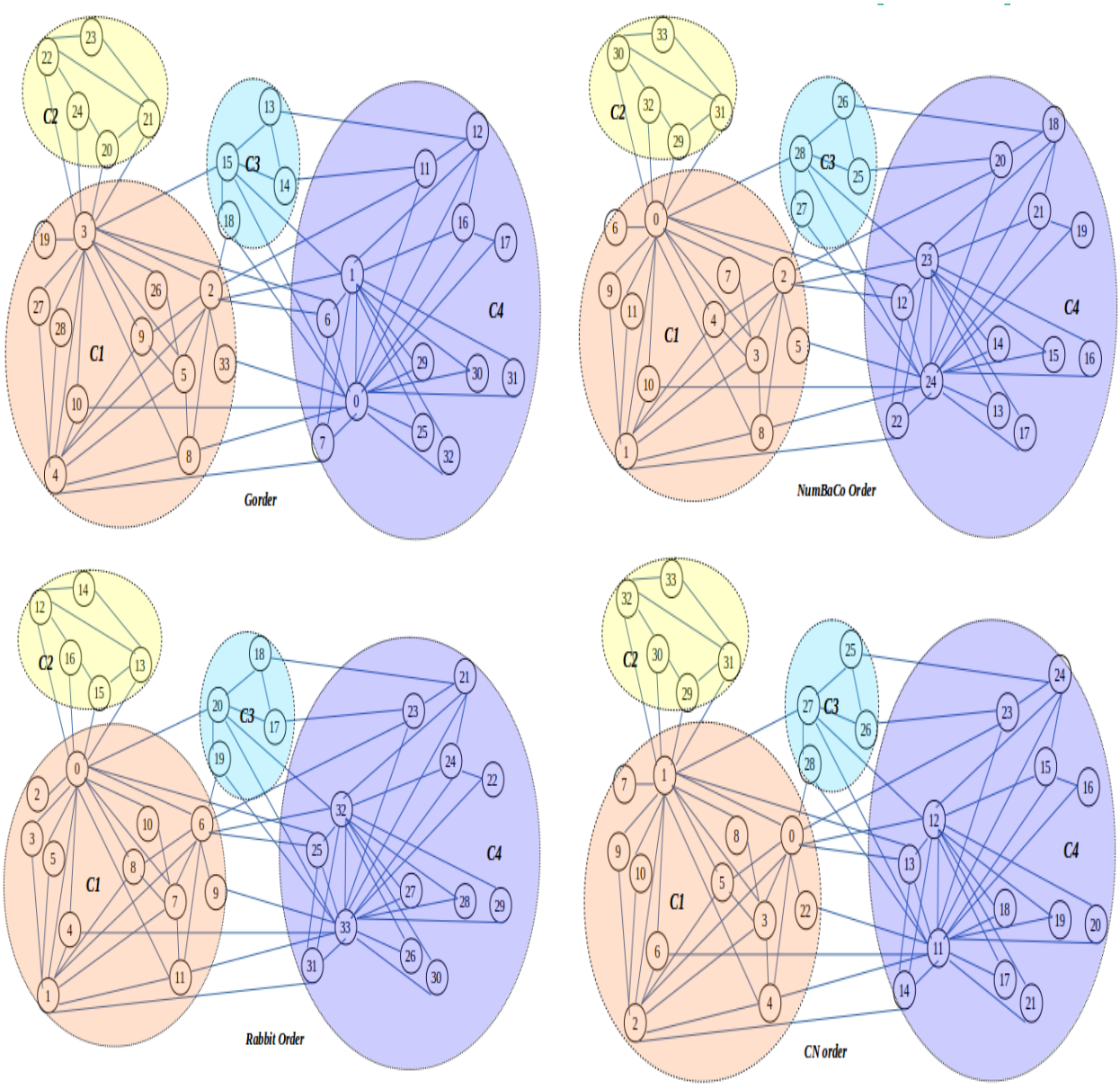


Figure A.3: karate graph with different orders

Appendix B

Katz with Green-Marl and Galois

B.1 Katz Score code with Green-Marl

Main portion Katz score code in Green-Marl syntax.

```
22  //Second step, with indirect neighbors, update path lengths and katz_score
23  While(l<=max_l){
24      Foreach(y:neighbor_set_1.Items){
25        Foreach(z:y.Nbrs){
26            If(neighbor_set_2.Has(z))
27              map_path_len_2[z] = map_path_len_2[z] + map_path_len_1[y];
28            Else
29            {
30                    map_path_len_2[z] = map_path_len_1[y];
31                    neighbor_set_2.Add(z);
32            }
33        }
34      }
35      Foreach(t:neighbor_set_2.Items)(!direct_neighbor.Has(t) && t!=x){
36              map_katz[t] = map_katz[t] + Pow(beta,l)*map_path_len_2[t];
37              all_reacheable_neighbor.Add(t);
38      }
39      neighbor_set_1 = neighbor_set_2;
40      neighbor_set_2 = temp_set;
41      l++;
42      i = 0; len_map_1 = map_path_len_1.Size();
43      len_map_2 = map_path_len_2.Size();
44      While(i < len_map_1)
45      {
46          map_path_len_1.Remove(map_path_len_1.GetMinKey());
47          i++;
48      }
49      i = 0;
50      While(i < len_map_2)
51      {
52          key = map_path_len_2.GetMinKey();
53          map_path_len_1[key] = map_path_len_2[key];
54          map_path_len_2.Remove(map_path_len_2.GetMinKey());
55          i++;
56      }
57  }
```

Main portion Katz score C++ code generated by Green-Marl Compiler.

```
42    while (l <= max_l)
43    {
44        i = 0 ;
45        len_map_1 = map_path_len_1.size() ;
46        gm_node_set::seq_iter y0_I = neighbor_set_1.prepare_seq_iteration();
47        while (y0_I.has_next())
48        {
49            node_t y0 = y0_I.get_next();
50            for (edge_t z_idx = G.begin[y0];z_idx < G.begin[y0+1] ; z_idx ++)
51            {
52                node_t z = G.node_idx [z_idx];
53                if (neighbor_set_2.is_in(z))
54                  map_path_len_2.setValue_seq(z, map_path_len_2.getValue(z)
                                                  + map_path_len_1.getValue(y0));
55                else
56                {
57                  map_path_len_2.setValue_seq(z,map_path_len_1.getValue(y0));
58                  neighbor_set_2.add_seq(z);
59                }
60            }
61        }
62        len_map_2 = map_path_len_2.size() ;
63        gm_node_set::seq_iter t_I = neighbor_set_2.prepare_seq_iteration();
64        while (t_I.has_next())
65        {
66         node_t t = t_I.get_next();
67         if ( !direct_neighbor.is_in(t) && (t != x))
68         {
69          map_katz.setValue_seq(t,(float)(map_katz.getValue(t)+ pow(beta, l)
                    * ((double)map_path_len_2.getValue(t))));
70          all_reacheable_neighbor.add_seq(t);
71         }
72        }
73        neighbor_set_1 = neighbor_set_2 ;
74        neighbor_set_2 = temp_set ;
75        l = l + 1 ;
76        while (i < len_map_1)
77        {
78            map_path_len_1.removeKey_seq(map_path_len_1.getMinKey_seq());
79            i = i + 1 ;
80        }
81        i = 0 ;
82        while (i < len_map_2)
83        {
84            key = map_path_len_2.getMinKey_seq() ;
85            map_path_len_1.setValue_seq(key, map_path_len_2.getValue(key));
86            map_path_len_2.removeKey_seq(map_path_len_2.getMinKey_seq());
87            i = i + 1 ;
88        }
89    }
```

B.2 Katz Score Code With Galois

```
void katz_score_node(Graph& G, float beta,  int32_t max_l, int32_t x_id)
{
  path_table_t path_table;
  float ktz_scr;
  int *removable_nodes = 0;
  Node_list_t *y_list = 0;

  path_table.init(max_l);
  path_table.build_path_table(x_id, G, max_l);
  removable_nodes = path_table.get_neighbors(x_id,G);
  path_table.extract_y_list(max_l, x_id, removable_nodes, &y_list);
  while(y_list!=0)
  {
              ktz_scr  = path_table.katz_score(y_list->id, beta, max_l);
              std::cout<<x_id<<" "<<y_list->id<<" "<<ktz_scr<<std::endl
              y_list   = y_list->suivant;
  }

  path_table.destroy_path_table();
  return;
}
```

Appendix C

Extended Abstract

C.1 Motivation and Goal

A complex network is a set made of a large number of entities interconnected with links. Social networks are an example of complex networks where entities are individuals and links are relationship (friendship, message passing or other) between these individuals. Complex networks are modeled by graphs where nodes represent entities and edges between nodes represent links between entities. Graphs that represent complex networks are usually big with sometime millions of nodes and millions or even billions of edges. For example at the end of the year 2017, Facebook social network boasts 2.13 billion active users per month and Twitter social network boasts 330 millions active users per month. This represents large volumes of data and leads some graph analysis applications to have long execution times.

Motivating example. In order to have an idea of the impact of graph size in the execution time of a graph analysis application, let take Katz score application running on four datasets: Karate (a social network of members of a karate club in United States), Quant-ph (a social network of researchers in quantum physic), DBLP (a social network of scientific article authors in computer science) and the social network Live Journal.

Katz score can be used for links prediction of a given graph. For a non-oriented graph with n nodes and m edges, the number of links to predict is $\frac{n(n-1)}{2} - 2m$. Let t be the time necessary to predict a link between two nodes with Katz score. Naively, i.e without any strategy to reduce time, the execution time taken by Katz score application is given by $T = (\frac{n(n-1)}{2} - 2m)t$. With Karate graph which has 35 nodes and 79 links $T_{karate} = (\frac{35*34}{2} - 2 * 79)t = 516t$. With Quant-ph graph which has 1060 nodes and 1044

edges, $T_{quant-ph} = 5.59 * 10^5 t$. With DBLP graph which has 195310 nodes and 2099732 edges, $T_{dblp} = 1.9 * 10^{10} t$. With Live Journal graph which has 3997962 nodes and 34681189 edges, $T_{liveJ} = 7.99 * 10^{12} t$.

In [34], the best implementation of Katz score algorithm with Karate dataset took $10.496s$. So an approximated value of t for Karate dataset is $t = \frac{T_{karate}}{516} = 2.03 * 10^{-2} s$. We assume that t has the same value for all the datasets. — But in practice, it is not really the case. Indeed, t value depends also of the graph size that influences cache misses of the application and hence the execution time. — Therefore, with $t = 2.03 * 10^{-2} s$:

$$\begin{cases} T_{karate} & = 10.496 \text{ seconds} \\ T_{quant-ph} & = 1.13 * 10^4 \text{ seconds} = 3.16 \text{ hours} \\ T_{dblp} & = 1.07 * 10^5 \text{ hours} = 12.29 \text{ years} \\ T_{liveJ} & = 5.15 * 10^3 \text{ years} \end{cases}$$

This example shows that, without taking into account the size of graphs in applications coming from complex networks analysis, the execution time can be very long (many hours or even years of execution if nothing is done). One way to reduce the execution time is using approximate solutions: rather than finding an exact calculation, some authors look for an approximate calculation that takes less time. It is the case with Paolo *et al.* [50] who choose a limited number of graph nodes to calculate an approximate value of *Betweenness Centrality*. In order to have an exact solution in a reasonable time, another one can use parallelism through high performance computers. However, the programmer may have difficulties to write efficient parallel programs that run on these architectures because he often finds himself writing low level platform specific code.

Before highlighting the goals and the contributions of this thesis, we will first present complex network properties, challenges in complex networks analysis and existing complex networks analysis parallelization approaches.

Complex Networks Properties. In addition to the big size that characterizes graphs, complex networks exhibit many other properties that distinguish them to random networks [32]. Two of these properties are described below:

- *Heterogeneity of node degree.* The degree of a node in a graph is the number of nodes connected (with an edge) to that node. The heterogeneity of node degree is characterized by the fact that there are (usually a small number of) nodes with higher degree compared to other nodes with smaller degree. In social networks nodes with higher degree are famous nodes.

- *Community structure*: A graph has a community structure if it is characterized by groups of nodes that have a higher density of edges within them, and a lower density of edges between other groups.

We argue in this thesis that a proper exploitation of these properties coupled with a proper exploitation of the target architectures can allow to reduce execution time of graph analysis applications.

Challenges in Complex Networks Analysis. The study of graphs is not recent. It started in 1736 with the bridges of Königsberg problem [15]. But since the advent of the web a few decades ago, graph analysis gains more interest. One reason of this interest is the huge among of data generated by complex networks (in particular social networks). This huge among of data requires high performance computers to be analyzed in a reasonable time.

On the other hand, applications from complex networks analysis are generally based on the exploration of the underlying graph. This exploration is most often local: after treating a node, the next nodes to be referenced belong to the neighborhood of that node. Since the underlying graph is usually unstructured, data access patterns of this applications tend to have poor locality. In addition, complex networks analysis applications are often irregular, i.e, all the computations required by each node in the application are not usually well known *a priori*. In the case of parallel (multi-threaded) applications, this irregularity can lead to an unbalance load among the workers (threads).

All those characteristics of complex networks applications — i.e. large amount of data to analyze, irregularity of applications, poor locality — give rise to many challenges that can be summarized in three mains: capacity, implementation and performance.

- **Capacity Challenge.** The capacity challenge refers to the fact that a dataset of a complex network application does not always fit into a single physical memory. In this thesis, we do not try to meet the capacity challenge. We consider the case where a dataset fits into a single physical memory, i.e simple memory machines, shared memory machines or NUMA (Non Uniform Memory Access) machines. We do not consider the case of distributed memory machines.

- **Implementation Challenge.** The implementation challenge is about writing easily a graph application from a given graph algorithm with a given programming language. Implementation challenge can be met with a programming language that is easy to use like Python or R programming languages. This challenge can also be met with a language that has its syntax close to graph analysis domain.

- **Performance Challenge.** In the performance challenge, we are looking for an efficient implementation of a given graph algorithm. An efficient implementation is the one that has the least execution time. This challenge is particularly difficult to met because of the irregularity and the poor locality of graph analysis applications. We think we can meet this challenge with an efficient parallelism ensured by a proper exploitation of complex network properties.

  Parallelism is not new in complex network analysis. In the next paragraph, we detail existing approaches in parallel complex networks analysis.

Parallelism in Complex Network Analysis. There are two main approaches to do parallelism in graph analysis: using programming models or using DSLs.

- **Using Programming Models.** Starting with a reference algorithm (assumed to be sequential), there are three steps that lead to a parallel program.

  Step 1) After designing his new algorithm, a complex networks analyzer needs to perform some experimental validation. That is, he wants to know if the implementation of his algorithm in a programming language produces the right results for given inputs. In other words, the complex networks analyzer is looking for a programming language that allows him to make a prototype in order to validate his algorithm. For this purpose, he uses programming languages like Python or R, which are known for their ease of use.

  Step 2) After the prototyping step, the complex networks analyzer is now looking for a language that allows him to study performance issues of its algorithm. Programming languages like C or C++ are often used for this.

  Step 3) At this step, the complex networks analyzer wants to parallelize his program in order to get more performances. In this case, he uses one of the existing programming models such as CUDA, Posix threads, MPI or OpenMP. For example, community detection is implemented with OpenMP in [38] or with CUDA in [46].

  It is not easy for the complex networks analyzer to follow all the three steps. This is because in addition to master the syntaxes of the used languages (Python or R, C or C++), it requires expertise in parallelism. There is another alternative that does not need so much efforts but that also ensures parallel graph analysis: parallel DSLs.

- **Using Parallel DSLs.** A Domain-Specific Language (DSL) is defined as a language designed for a precise domain (for example SQL) as opposed to a general purpose language (for example Java, C). A DSL is either stand alone (has its own compiler) or embedded in an existing general purpose language.

  Parallel DSLs in graph analysis are mobilizing more and more scientific community. For example, Hassan Chafi *et al.* [9] propose a generic DSL intended to be used by parallel DSL designers; Green-Marl [19] is a stand-alone DSL developed for parallel graph analysis; Galois [33] is another DSL designed for graph analysis, but it is an embedded one.

  When using parallel graph DSLs, the only effort asked to a complex network analyzer is to learn the DSL syntax. All the efforts needed to get an efficient implementation are transparent to him.

Goal and Methodology. The general question that emerges is: how to develop graph analysis applications that are both *easy to write* (implementation challenge) and *efficient* (performance challenge)? We try to answer this question with parallelism and also with complex networks properties.

We saw that we can do parallel graph analysis with programming models or with DSLs. While the first approach leads to efficient graph applications (performance challenge), the second targets both the performance and the implementation challenges. In this thesis, we choose the second approach. We are looking for a DSL that ensures easy and efficient complex networks analysis.

There are many DSLs designed for graph analysis. But all those DSLs were designed to implement classical graph algorithms. In that way, the implementation of the new algorithms arisen by complex networks with these DSLs is not always efficient. We argue in this thesis that one way to ensure this implementation to be efficient is by exploiting complex network properties. So the wanted DSL may exploit these properties in order to produce efficient implementation. Since none of the existing DSLs does not exploit these properties, we have two options: build a new DSL or use existing one. We choose the second option. Our methodology implies:

1. Exploit some complex networks properties in order to provide heuristics that improve graph applications performances.

2. Then, implement these heuristics in existing graph DSLs (Galois and Green-Marl).

C.2 Contribution

Our contribution in this thesis has three aspects: – Graph Ordering Heuristics, – Degree-aware Scheduling Heuristics, – and Heuristics Integration in Graph DSLs.

Graph Ordering Heuristics. The first contribution of this thesis showed the exploitation of community structure in order to design community-aware graph ordering heuristics for cache misses reduction. After formalizing the Numbering Graph Problem (NGP) for cache misses reduction as linear arrangement problem (a well known NP-Complete problem), we proposed an heuristic called NumBaCo to solve that problem. Then, we compared Numbaco to Gorder and Rabbit (which appeared in the literature at the same period NumBaCo was proposed). This comparison allowed to design Cn-order, another heuristic that combines advantages of the three algorithms (Gorder, Rabbit and NumBaCo) to solve the NGP. Experimental results with one thread on Core2, Numa4 and Numa24 (with Pagerank and livejournal for example) showed that cache misses reduction and execution time reduction was ensured and also, that Cn-order uses well the advantages of the other orders and outperforms them.

Degree-aware Scheduling Heuristics. The previous analysis stayed at only one thread. The second contribution this thesis considered the case of multiple threads applications. In that case, cache misses reduction is not sufficient to ensure execution time reduction; one should also take into account load balancing among threads. In that way, heterogeneity of node degree was used in order to design Deg-scheduling, a heuristic to solve degree-aware scheduling problem (formalized as the NP-Complete Knapsack problem). Deg-scheduling was combined to Cn-order, NumBaCo, Rabbit, and Gorder to form respectively Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling.

Experimental results with many threads on Numa4 (with Pagerank and livejournal for example) showed that Degree-aware scheduling heuristics (Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling and Gor-deg-scheduling) outperform their homologous graph ordering heuristics (Cn-order, NumBaCo, Rabbit, and Gorder) when they are compared two by two. This is because degree-aware scheduling heuristics ensure both cache misses reduction and load balancing among threads, while graph ordering heuristics ensure only cache misses reduction.

Heuristics integration in Graph DSLs. The last contribution of this thesis the implementation of graph ordering heuristics and degree-aware scheduling heuristics parallel graph DSLS. After proposing a general methodology of heuristics integration in parallel graph DSLs, we implement the proposed heuristics into Green-Marl and Galois. Experimental results showed that with Green-Marl, performances are increased by both graph ordering heuristics and degree-aware scheduling heuristics (time was reduced by 35% due to heuristics). But with Galois, performances are increased only with graph ordering heuristics (time was reduced by 48% due to heuristics).

Publications. This thesis leads to three publications with reading committee (2 conferences and 1 journal):

1. Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Using Complex-Network properties For Efficient Graph Analysis.* Accepted for publication in Proceedings of International Conference on Parallel Computing, ParCo 2017, Bologna, Italy 12-15 september 2017. (Proceedings will appear in IOS Press, Amsterdam in March/April 2018 and will be listed by **Scopus**)

2. Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Social network ordering based on communities to reduce cache misses.* Revue ARIMA, Volume 24 - 2016-2017 - Special issue CRI 2015, May 2017. (Arima journal is indexed listed by **MathSciNet**)

3. Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Exploitation de la structure en communautés pour la réduction de défauts de cache dans la fouille des réseaux sociaux.* In : Conférence de Recherche en Informatique (CRI). 2015.

This thesis is an extension of our master thesis which leads to two publications with reading committee.

4. Marcio Castro, Emilio Francesquini, Thomas Messi Nguélé, and Jean-François Mehaut. *Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Applications.* In Proceedings of the IA^3 Workshop on Irregular Applications : Architectures & Algorithms, hold in the SC conference, Denver, US, nov 2013. ACM. Accepted.

5. Marcio Castro, Emilio Francesquini, Thomas Messi Nguélé, and Jean-François Méhaut. *Multicœurs et manycœurs : Une analyse de la performance et l'Éfficacité Énergétique d'une application irrégulière.* Accepted for presentation in CRI'2013, Conférence de Recherche en Informatique, Yaoundé Décembre 2013.

C.3 Perspectives

The results gotten in this thesis confirm that the performance of complex network analysis can be improved thanks to a proper parallelism combined to a proper exploitation of some complex network properties. This thesis also confirms that, instead of building a new DSL for complex network analysis, one can improve the existing graph DSLs. This improvement is done by integrating into these DSLs, heuristics that can make the applications implemented with them faster. However, the work presented in this thesis is still improvable. In the next paragraphs, we will present perspectives that follow directly this thesis.

General Perspectives. In this thesis we use two complex networks properties community structure and heterogeneity of node degree. Others properties like small word effect or transitivity can also be studied to design heuristics that improve graph applications performance. Instead of using complex networks properties to design heuristics, one can imagine to use machine learning.

Another perspective concerns the theoretical aspect of this thesis. Remember we showed that graph ordering for cache misses reduction and degree-aware scheduling for load balancing problems are NP-complete. We provided heuristics to solve them. But we didn't show how far these heuristics are to the optimal solutions. It is good to know it in the nearest future.

Perspectives in Graph Ordering Heuristics. The first contribution of this thesis focuses on graph ordering for cache misses reduction. In order to design new heuristics, one can compare theoretically and experimentally graph ordering for cache misses reduction with graph ordering for compression.

We also mentioned that Gorder can be improved by using item-set strategy (introduced by Agrawal *et al.* [1]). And even NumBaCo and Rabbit can also be improved:

- By using local communities detection to assign numbers inside the community.
- By using other communities detection algorithms (we used GCA and Louvain).

Perspectives in Heuristics Integration. We showed that with Galois DSL, we have no gain with scheduling heuristics. To take advantage to our scheduling heuristics, one should design a new algorithm that takes advantage of both internal scheduler of Galois and our degree-aware schedulers.

Another work to do in a near future is to implement these heuristics in other existing graph analysis platforms (different to Galois and Green-Marl). That will probably contribute to increase their performances.

Appendix D

Résumé Étendu

D.1 Motivation et Objectif

Un réseau complexe est un ensemble constitué d'un grand nombre d'entités interconnectés par des liens. Les réseaux sociaux sont un exemple de réseaux complexes où les entités sont des individus et les liens sont les relations (d'amitié, d'échange de messages) entre ces individus. Les réseaux complexes sont modélisés par des graphes dans lesquels les noeuds représentent les entités et les arêtes entre les noeuds représentent les liens entre ces entités.

Les graphes issus des réseaux complexes sont souvent très grands (des millions de noeuds avec des millions d'arêtes voire des milliards d'arêtes). Par exemple, le réseau social Facebook revendique 2,13 milliards d'utilisateurs actifs par mois en fin d'année 2017; le réseau social Twitter revendique 330 millions d'utilisateurs actifs par mois à la même période. Ceci représente de très gros volumes de données pouvant générer de longues durées d'exécution lors de l'analyse.

Exemple Introductif. Pour se faire une idée de l'impact de la taille du graphe dans l'analyse des graphes issus des réseaux complexes, prenons l'exemple du score de Katz qui s'exécute sur quatre jeux de données de tailles différentes: Karate (un réseau social des membres d'un club de karate d'une université aux États Unis), Quant-ph (le réseau des chercheurs de la physique quantique), DBLP (le réseau des auteurs des articles scientiques en informatique) et le réseau social Live Journal.

Le score de Katz peut être utilisé pour la prédiction des liens entre les noeuds d'un graphe. Pour un graphe (non orienté) de n noeuds et m arêtes, le nombre de liens à prédire est de $\frac{n(n-1)}{2} - 2m$. Ainsi, si l'on suppose que le coût (le temps de calcul) du

score de Katz permettant de prédire un lien entre deux noeuds est t, de façon naïve (sans aucune stratégie de réduction du coût), le temps d'exécution de l'algorithme est $T = (\frac{n(n-1)}{2} - 2m)t$. Avec le graphe Karate qui a 35 noeuds et 79 liens, $T_{karate} = (\frac{35*34}{2} - 2*79)t = 516t$. Avec Quant-ph qui a 1060 noeuds et 1044 arête, $T_{quant-ph} = 5.59 * 10^5 t$. Avec DBLP qui a 195310 noeuds et 2099732 arêtes, $T_{dblp} = 1.9 * 10^{10} t$. Et enfin, avec le graphe Live Journal qui a 3997962 noeuds et 34681189 arêtes, $T_{liveJ} = 7.99 * 10^{12} t$.

Dans notre mémoire de master [34], l'exécution de la meilleur implémentation du score de Katz avec le jeu de données Karate avait pris $10.496s$. Ainsi, une valeur approchée de t pour le jeu de données Karate est $t = \frac{T_{karate}}{516} = 2.03 * 10^{-2} s$. On suppose que t a la même valeur pour tous les jeux de données. — Ceci n'est pas le cas en pratique, car la valeur de t dépend aussi de la taille du graphe qui influence les défauts de cache et par là le temps d'exécution. — Pour $t = 2.03 * 10^{-2} s$:

$$\begin{cases} T_{karate} & = 10.496 \text{ secondes} \\ T_{quant-ph} & = 1.13 * 10^4 \text{ secondes} = 3.16 \text{ heures} \\ T_{dblp} & = 1.07 * 10^5 \text{ heures} = 12.29 \text{ années} \\ T_{liveJ} & = 5.15 * 10^3 \text{ années} \end{cases}$$

Cet exemple montre que sans une prise en compte de la taille des graphes dans les algorithmes issus des réseaux complexes, le temps d'exécution peut être très long (sur l'exemple, on voit qu'on peut atteindre plusieurs heures, voire plusieurs années d'exécution si rien n'est fait). Une façon de réduire ce temps d'exécution est l'usage des solutions approchées: plutôt que d'effectuer un calcul exact, certains auteurs préconisent d'effectuer un calcul approché destiné à prendre un court temps d'exécution. C'est le cas de Paolo *et al.* [50] qui ont utilisé les communautés pour choisir un nombre limité des noeuds du graphe pour un calcul approché du *Betweenness Centrality*. Pour avoir une solution exacte en un temps raisonnable, une autre façon de procéder est d'utiliser le parallélisme à travers les machines hautement performante. Toutefois, le programmeur peut avoir des difficultés à écrire des programmes parallèles d'analyse de graphe efficaces tournant sur ces architectures, car il se retrouve souvent à écrire un code de bas niveau spécifique à la plate-forme.

Avant de mettre en évidence l'objectif et la contribution de cette thèse, nous allons d'abord présenter les propriétés des réseaux complexes, les défis dans l'analyse des réseaux complexes et les approches existantes de parallisation des applications d'analyse de graphes.

Les Propriétés des Réseaux Complexes. En plus de la grande taille qui caractérise les graphes, les réseaux complexes ont d'autres propriétés qui les distinguent des réseaux aléatoires [32]. Deux de ces propriétés sont décrites ci-après:

- *L'hétérogénéité des dégrés des noeuds.* Le dégré d'un noeud dans un graphe est le nombre de noeuds connectés à ce noeud (par une arête). L'hétérogénéité des dégrés des noeuds se caractérise par un petit nombre de noeuds ayant de très grands dégrés et un grand nombre de noeuds ayant de petits dégrés. Dans les réseaux sociaux, ce sont les célébrités qui ont souvent des dégrés très élévés.

- *La structure en communautés*: Un graphe a une strucutre en communautés s'il contient des groupes ayant une forte densité de liens à l'intérieur (des groupes) et une faible densité de liens à l'extérieurs des groupes.

Nous argumentons dans cette thèse qu'une bonne exploitation des propriétes ainsi présentées couplée à une bonne exploitation des architectures d'exécutions cibles permet de réduire le temps d'exécution des applications d'analyse des réseaux complexes.

Les défis dans l'analyse des réseaux complexes. L'étude des graphes n'est pas récente. Elle date de 1736 avec le problème des ponts de Königsberg [15]. C'est l'avènement du web il y a quelques décennies qui augmente l'intérêt porté sur l'analyse des graphes. Une des principales raisons de cet intérêt est la grande quantité des données générées par les réseaux complexes (et en particulier les réseaux sociaux). Cette grande quantité des données nécessite des ordinateurs hautement performants permettant de les analyser en temps raisonnable.

D'autre part, les applications issues de l'analyse des réseaux complexes sont généralement basées sur l'exploration du graphe sous-jacent. Cette exploration est souvent locale: après avoir traité un noeud, les prochains noeuds auquels l'application fait reférence appartiennent au voisinage de ce noeud. Étant donné que le graphe sous-jacent est habituellement non structuré, les séquences d'accès aux données en mémoire tendent à avoir une faible localité. En plus, les applications issues de l'analyse des réseaux complexes sont souvent irrégulières, c'est à dire que l'ensemble et le nombre de calculs réquis pour chaque noeud du graphe n'est pas souvent connu d'avance. Dans le cas des applications parallèles (multithreadées), cette irrégularité peut conduire à un déséquilibre de charges entre les threads.

Toutes les caractéristiques des applications issues de l'analyse des réseaux complexes – à savoir l'irrégularité, la grande quantité des données à analyser, la faible localité –

donnent lieu à de nombreux défis dont les principaux sont résumés en trois points: la capacité, l'implémentation et la performance.

- **Le Défis de la Capacité.** Il fait référence au fait qu'un jeu de donnée d'une application d'analyse des réseaux complexes ne suffit pas toujours sur une machine ayant une seule mémoire physique. Dans cette thèse, nous ne cherchons pas à relever ce défis, nous considérons le cas des jeux de données pouvant suffir sur des machines à mémoire partagées ou des machines NUMA (Non Uniform Memory Access) machines. Nous ne prenons pas en compte le cas des machines à mémoire distribuée.

- **Le Défis de l'Implémentation.** Il s'agit ici de pouvoir écrire facilement une application à partir d'un algorithme de graphe et avec un langage de programmation donné. Ce défis peut être relevé avec un langage facile à utiliser comme Python ou R. Ce défis peut aussi se relever avec un langage dont la syntaxe est proche du domaine de l'analyse des graphes.

- **Le Défis de la Performance.** Ici nous recherchons une implémentation efficace à partir d'un algorithme de graphe donné. Une implémentation est dite efficace si elle donne le plus petit temps d'exécution. C'est un défis particulièrement difficile à relever à de l'irrégularité et de la faible localité des applications issues de l'anlyse des réseaux complexes. Nous pensons pouvoir relever ce défis en nous servant d'un parallélisme efficace assuré par une exploitation propice des propriétés des réseaux complexes.

  L'usage du parallélisme n'est pas nouveau dans l'analyse des réseaux complexes. Le paragraphe suivant rappelle les approches existante d'analyse parallèle des réseaux complexes.

Parallélisme dans l'Analyse des Réseaux Complexes. Il y a deux principales façons de paralléliser les applications d'analyse de graphe: en utilisant les modèles de programmation parallèle ou en utilisant les DSLs parallèles de graphe.

- **Usage des modèles de programmation.** À partir d'un algorithme (supposé séquentiel), il y a trois étapes qui conduisent à un programme parallèle:

  Étape 1) Après avoir conçu son nouvel algorithme, l'algorithmicien (des réseaux complexes) voudrait le valider à travers des expériementations. En d'autres termes, il veut savoir si l'implémentation de son algorithme dans un

langage de programmation produit des résultats correctes à partir des données entrées. L'algorithmicien est à la recherche d'un langage de programmation qui lui permet de faire un prototype afin de valider son algorithme. Pour cette raison, il se sert des langages comme Python ou R qui sontconnus comme étant facile à utiliser.

Étape 2) Après l'étapte de prototypage, l'algorithmicien se met à la recherche d'un langage qui lui permet d'étudier les performances de son algorithme. Les langages de programmation comme C ou C++ sont bien indiqués dans ce cas.

Étape 3) À cet étape, l'algorithmicien veut paralléliser son programme pour augmenter les performances. Il utilise dans ce cas un des modèles de programmation existant comme CUDA, Posix threads, MPI ou OpenMP. Par exemple, l'algorithme de détection des communautés a été implémenté avec OpenMP dans [38] ou avec CUDA dans [46].

Ce n'est pas facile pour l'algorithmicien de suivre toutes ces étapes. En effet, en plus des langages de programmation utilisés (Python ou R, C ou C++), il doit avoir de l'expertise en parallélisme. Une autre alternative pouvant être adoptée pour l'analyse parallèle des graphes est l'usage des DSLs.

- **Usage des DSLs parallèles de graphe.** Un DSL (Domain-Specific Language) est un langage conçu pour un domaine précis (exemple SQL pour les requêtes aux bases de données) en opposition à un langage généraliste (comme Java ou C). Un DSL est soit "stand alone" (il a son propre compilateur), soit embarqué (dans un langage généraliste).

  Les DSLs parallèles d'analyse de graphe mobilisent de plus en plus la communauté scientifique. Par exemple, Hassan Chafi *et al.* [9] proposent un DSL générique destiné à être utilisé par les développeurs de DSLs parallèles; Green-Marl [19] est un DSL "stand-alone" developpé pour l'analyse parallèle des graphes; Galois [33] est aussi un DSL conçu pour l'analyse des graphes mais plutôt embarqué.

  Quand on utilise les DSLs parallèles de graphe, le seul effort à fournir par l'algorithmicien est de maitriser la syntaxe du DSL. Tous les autres efforts permettant d'avoir une implémentation efficace lui sont transparents.

Objectif de la Thèse et Méthodologie. La question générale que l'on se pose dans cette thèse est celle de savoir: comment développer les applications d'analyse des graphes qui sont à la fois facile à écrire (défis d'implémentation) et efficace (défis de performance)?

Nous essayons de répondre à cette question en nous servant du parallélisme et des propriétés des réseaux complexes.

Nous avons vu que l'analyse parallèle des graphes se fait en utilisant les modèles de programmation ou en utilisant les DSLs. Si la première approche permet d'aboutir à une application d'analyse de graphes efficace (relevant ainsi le défis de performance), la seconde approche permet de relever à la fois le défis de performance et le défis d'implémentation. Dans cette thèse, nous choisissons la seconde approche. Nous sommes à la recherche d'un DSL permettant d'analyser facilement et efficacement les réseaux complexes. Il existe plusieurs DSLs conçus pour l'analyse des graphes. Mais tous ces DSLs ont été conçus pour les algorithmes classiques des graphes; ces DSLs ne permettent pas toujours d'avoir une implémentation efficace des algorithmes issus des réseaux complexes.

Cette thèse argumente qu'une façon de s'assurer une implémentation efficace est d'exploiter les propriétés des réseaux complexes. Ainsi, le DSL recherché devra pouvoir exploiter ces propriétés pour produire une implémentation efficace. Étant donné qu'aucun des DSLs existant ne permet d'exploiter ces propriétés, nous avons deux options pour atteindre notre objectif: construire un nouveau DSL ou alors modifier un qui existe déjà. Nous avons choisi la deuxième option. Notre méthodologie implique deux étapes:

1. Exploiter les propriétés des réseaux complexes afin de produire des heuristiques permettant d'améliorer les performances des applications d'analyse de graphes.

2. Ensuite, implémenter ces heuristiques dans des DSLs de graphes existant (Galois et Green-Marl).

D.2 Contribution

Notre contribution dans cette thèse se présente sous trois aspects: – les heuristiques de numérotation des graphes, – les heuristiques d'ordonnancement, – l'intégration des heuristiques dans les DSLs de graphes.

Heuristiques de Numérotation des Graphes. La première contribution de cette thèse met en évidence l'exploitation de la structure en communautés des réseaux complexes pour la conception des algorithmes de numérotation des graphes permettant la réduction des défauts de cache des applications tournant dans ces graphes. Après avoir formalisé le problème de numérotation des réseaux pour la réduction des défauts de cache comme un problème d'arrangement linéaire optimal (un problème NP-complet bien

connu), nous avons proposé NumBaCo pour le résoudre. Nous avons comparé NumBaCo à Gorder et Rabbit (qui ont apparu dans la littérature en même temps que NumBaCo). Cette comparaison nous a permis de développer CN-order, une autres heuristique qui combine les avantages des trois algorithmes (Gorder, Rabbit, NumBaCo).

Les résultats expérimentaux sur un thread avec les architectures Core2, Numa4 et Numa24 ont montré que les défauts de cache et ensuite le temps d'exécution étaient effectivement réduits; et que CN-order se sert bien des avantages des autres heuristiques de numérotation pour produire les meilleurs résultats.

Heuristiques d'ordonnancement. La deuxième contribution de cette thèse a considéré le cas des applications multi-threadées. Dans ce cas, la réduction des défauts de cache n'est pas suffisante pour assurer la diminution du temps d'exécution; l'équilibre des charges entre les threads doit être assuré pour éviter que certains threads prennent du retard et ralentissent ainsi toute l'application. Dans ce sens, nous nous sommes servis de la propriéte de l'hétérogénéité des dégrés des noeuds pour développer Deg-scheduling, une heuristique permettant de résoudre le problème d'ordonnancement basé sur les dégrés (un poblème formalisé comme le problème du sac à dos multiple). Deg-scheduling a été combiné à CN-order, NumBaCo, Rabbit, et Gorder pour construire respectivement Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling et Gor-deg-scheduling.

Les résultats expérimentaux avec plusieurs threads sur l'architecture Numa4 (avec Pagerank et Live Journal par exemple) montrent que les heuristiques d'ordonnancement (Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling et Gor-deg-scheduling) produisent de meilleurs résultats que leurs homologues des heuristiques de numérotation (CN-order, NumBaCo, Rabbit, and Gorder) lorsqu'on les compare deux à deux. Ces résultats s'expliquent par le fait les heuristiques d'ordonnancement assurent à la fois la réduction des défauts de cache et l'équilibre des des charges entre les threads, tandisque les heuristiques de numérotation assurent uniquement la réduction des défauts de cache.

Intégration des Heuristiques dans les DSLs de Graphes. La dernière contribution de cette thèse porte sur l'intégration des deux catégories d'heuristiques développées dans les DSLs parallèles d'analyse des graphes. Après avoir proposé une méthodologie générale d'intégration des heuristiques, nous les avons implémentées dans les DSLs de graphe Green-Marl et Galois. Nous avons montré qu'avec Green-Marl, les performances sont améliorées à la fois grâce aux heuristiques de numérotation et grâce aux heuristiques d'ordonnancement (temps réduit de 35% grâce aux heuristiques). Mais avec Galois, les performances sont améliorées uniquement grâce aux heuristiques de numérotation (le

temps est réduit de 48% grâce à ces heuristiques).

Publications. Cette thèse a donné lieu à trois articles scientifiques avec commité de lecture (2 conférences et 1 journal):

1. Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Using Complex-Network properties For Efficient Graph Analysis.* Accepted for publication in Proceedings of International Conference on Parallel Computing, ParCo 2017, Bologna, Italy 12-15 september 2017. (Proceedings will appear in IOS Press, Amsterdam in March/April 2018 and will be listed by **Scopus**)

2. Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Social network ordering based on communities to reduce cache misses.* Revue ARIMA, Volume 24 - 2016-2017 - Special issue CRI 2015, May 2017. (Arima journal is indexed listed by **MathSciNet**)

3. Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Exploitation de la structure en communautés pour la réduction de défauts de cache dans la fouille des réseaux sociaux.* In : Conférence de Recherche en Informatique (CRI). 2015.

Cette thèse est une extension de notre mémoire de master qui avait donné lieu à deux publications avec commité de lecture.

4. Marcio Castro, Emilio Francesquini, Thomas Messi Nguélé, and Jean-François Mehaut. *Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Applications.* In Proceedings of the IA^3 Workshop on Irregular Applications : Architectures & Algorithms, hold in the SC conference, Denver, US, nov 2013. ACM. Accepted.

5. Marcio Castro, Emilio Francesquini, Thomas Messi Nguélé, and Jean-François Méhaut. *Multicœurs et manycœurs : Une analyse de la performance et l'Éfficacité Énergétique d'une application irrégulière.* Accepted for presentation in CRI'2013, Conférence de Recherche en Informatique, Yaoundé Décembre 2013.

D.3 Perspectives

Cette thèse montre que les performances des applications d'analyse des réseaux complexes peuvent être améliorées en utilisant le parallélisme et les propriétés des réseaux complexes. Elle montre également que plutôt que de créer un nouveau DSL pour l'analyse

des réseaux complexes, on peut améliorer les DSLs de graphes existant en y intégrant des heuristiques permettant de rendre plus rapides les applications issues de l'analyse des réseaux complexes et exprimées avec ces DSLs. Toutefois, le travail effectué dans cette thèse peut être amélioré ou étendu.

Perspectives Générales. Dans cette thèse, nous avons utilisé deux propriétés des réseaux complexes à savoir la structure en communautés des noeuds du graphe et l'hétérogénéité des dégrés des noeuds. D'autres propriétés comme l'effet petit monde ou la transitivité peuvent aussi être étudiées afin de développer des heuristiques permettant d'améliorer les performances des applications issues de l'analyse des réseaux complexes. Aussi, au lieu d'utiliser les propriétés des réseaux complexes pour développer les heuristiques, on peut imaginer l'usage des méthodes de machine learning.

Une autre perspective concerne l'aspect théorique de la thèse. Nous avons montré que le problème de numérotation des graphes pour la diminution des défauts de cache et le problème de l'ordonnancement basé sur les dégrés des noeuds pour l'équlibrage des charges sont NP-complets. Nous avons proposé des heuristiques pour les résoudre, mais nous n'avons pas montrer à quelle distance ces heuristiques se situent de la solution optimale. Ceci reste une question ouverte qui mérite d'être étudiée dans un futur très proche.

Perspectives dans les Heuristiques de Numérotation. La première contribution de cette thèse s'est faite sur la numérotation de graphe pour la réduction des défauts de cache. Afin de concevoir de nouvelles heuristiques, on peut comparer de façon théorique et de façon expérimentale les heuristiques de numérotation pour la réduction des défauts de cache avec les heuristiques de numérotation pour la compression des graphes.

Gorder peut être amélioré en utilisant la stratégie des item-sets (introduite par Agrawal *et al.* [1]). De même, NumBaCo et Rabbit peuvent être améliorés:

- En utilisant la détection des communautés locales pour assigner les numéros aux noeuds à l'intérieur des communautés.
- En utilisant des algorithmes de détection des communautés autres que GCA et Louvain (utilisé dans cette thèse).

Perspectives dans l'Intégration des Heuristiques dans les DSLs. Nous avons montré que pour le DSL Galois, nous n'avons pas de gain avec les heuristiques d'ordonnancement. Pour tirer profit des heuristiques d'ordonnancement proposées dans cette thèse, on doit

développer un algorithme capable de bénéficier de l'ordonnanceur interne de Galois et ceux que nous avons proposés.

Une autre perspective à réaliser dans un futur proche est l'intégration des heuristiques proposées dans d'autres plates-formes d'analyse de graphes. Ceci contribuera à accroitre leurs performances.

Appendix E

Rapport de Pré-Soutenance

Intitulé du mémoire: Langages Dédiés pour la Fouille des Réseaux Sociaux sur des Plates-formes Multi-coeurs. Présenté par **Thomas MESSI NGUELE** en vue de l'obtention du titre de Docteur/PhD en Informatique de l'Université de Yaoundé I et de Docteur en informatique de l'Université Grenoble Alpes

E.1 Problématique de la Thèse

Les travaux de **Thomas Messi Nguélé** portent sur les réseaux complexes c'est-à-dire des ensembles constitués d'un grand nombre d'entités interconnectées par des liens, et ont été réalisés dans le cadre d'une co-tutelle de l'Université de Yaoundé I et de l'Université Grenoble Alpes. Les réseaux complexes sont modélisés par des graphes dans lesquels les noeuds représentent les entités et les arêtes entre les noeuds représentent les liens entre ces entités. Ces graphes se caractérisent par un très grand nombre de sommets et une très faible densité de liensLes réseaux sociaux sont des exemples de réseaux complexes où les entités sont des individus et les liens sont les relations (d'amitié, d'échange de messages) entre ces individus. L'analyse des réseaux complexes est généralement basée sur l'exploration locale du graphe sous-jacent : après avoir traité un nœud u, les prochains noeuds auxquels l'application fait référence appartiennent au voisinage de u. Étant donné que le graphe sous-jacent est habituellement non structuré, les séquences d'accès aux données en mémoire tendent à avoir une faible localité lorsque qu'on utilise par exemple le stockage de Yale qui est l'un des meilleurs connus. En plus, dans les applications basées sur l'analyse des réseaux le nombre de calculs requis pour chaque noeud peut être très variable, ce qui, dans les mises en œuvre parallèles (multithreadées), se traduit par un déséquilibre de charges entre les threads. Le travail demandé à **Thomas Messi Nguélé** était était lié au développement d'applications d'analyse des réseaux sociaux, qui soient à la fois faciles à écrire et efficaces. A cet effet, **Thomas Messi Nguélé** a exploré deux pistes:

1. L'exploitation de la structure en communautés pour définir des techniques de stockage qui

réduisent les défauts de cache lors de l'analyse des réseaux sociaux

2. La prise en compte de l'hétérogénéité des degrés des nœuds pour optimiser la mise en œuvre parallèle

E.2 Contenu du mémoire

E.2.1 Chapitre 3

Dans ce chapitre qui contient la première contribution de la thèse, **Thomas Messi Nguélé** met en évidence l'exploitation de la structure en communautés des réseaux complexes pour la conception des algorithmes de numérotation des graphes permettant la réduction des défauts de cache. A cet effet, il propose une heuristique baptisée NumBaCo qu'il a ensuite comparée à Gorder et Rabbit (deux algorithmes publiés dans la littérature en même temps que NumBaCo). Cette comparaison lui permet de développer CN-order, une nouvelle heuristique qui combine les avantages des trois algorithmes (Gorder, Rabbit, NumBaCo). Les résultats expérimentaux, sur les architectures Core2, Numa4 et Numa24, ont montré que les défauts de cache et le temps d'exécution étaient effectivement réduits. De plus, CN-order combine effectivement les avantages des autres heuristiques de numérotation pour produire les meilleurs résultats.

Ce Chapitre a donné lieu à une communication à une conférence internationale et à un article dans une revue scientifique référencée dans MathSciNet (ARIMA).

- Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Social network ordering based on communities to reduce cache misses.* Revue ARIMA, Volume 24 - 2016-2017 - Special issue CRI 2015, May 2017. (Arima journal is indexed listed by MathSciNet)

- Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Exploitation de la structure en communautés pour la réduction de défauts de cache dans la fouille des réseaux sociaux.* In : Conférence de Recherche en Informatique (CRI). 2015.

E.2.2 Chapitre 4

La deuxième contribution de la thèse de **Thomas Messi Nguélé** concerne les applications multi-threadées. Dans ce cas, la réduction des défauts de cache n'est pas suffisante pour assurer la diminution du temps d'exécution. En effet, l'équilibre des charges entre les threads doit être assuré pour éviter que certains threads soient trop chargés, et ralentissent toute l'application. Pour ce faire, **Thomas Messi Nguélé** montre comment tenir compte de l'hétérogénéité des degrés des noeuds pour développer une heuristique baptisée Deg-scheduling. Cette technique est ensuite combinée avec CN-order, NumBaCo, Rabbit, et Gorder pour construire respectivement Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling et Gor-deg-scheduling.

Les résultats expérimentaux sur l'architecture Numa4 obtenus notamment avec l'algorithme classique Pagerank, montrent que les heuristiques d'ordonnancement Comm-deg-scheduling, Numb-deg-scheduling, Rab-deg-scheduling et Gor-deg-scheduling produisent de meilleurs résultats que leurs homologues du chapitre 2.

Ce travail a fait l'objet d'une présentation au collègue international ParCo2017, et dont le actes seront publiés par IO Press, avec indexation Scopus.

- Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. *Using Complex Network properties For Efficient Graph Analysis.* Accepted for presentation in Proceedings of International Conference on Parallel Computing, ParCo 2017, Bologna, Italy 12-15 September (to appear in IOS Press, Amsterdam in 2018).

E.2.3 Chapitres 5 et 6

Ces chapitres portent sur l'intégration des heuristiques développées aux chapitres précédents dans des environnements qui offrent des langages dédiés pour l'analyse des graphes (DSL en anglais). Avec Green-Marl, les performances sont améliorées à la fois grâce aux heuristiques de numérotation et grâce aux heuristiques d'ordonnancement (temps réduit de 35

E.3 Conclusion

Thomas Messi Nguélé présente pour la thèse un ensemble de travaux sur le stockage des grands réseaux sociaux en vue d'une exécution performante des algorithmes d'analyse sur plates formes multi-cœurs. Il propose à cet effet d'exploiter la structure en communautés pour réduire les défauts de cache et de tenir compte de l'hétérogénéité des degrés des nœuds pour équilibrer les threads. Les résultats expérimentaux confirment les bonnes performances de ces approches.

Les publications dans des conférences internationales qui font autorité dans le domaine et dans une revue internationale indexée MathSciNet atteste de la qualité des résultats.

Pour toutes ces raisons, nous émettons un avis très favorable pour la soutenance de cette thèse présentée par Monsieur **Thomas Messi Nguélé** en vue du Doctorat/PhD en informatique de l'Université de Yaoundé I et de Docteur en informatique de l'Université de Grenoble Alpes./.

Pr Jean-Francois Méhaut (UGA) *Pr Maurice Tchuente (UY1)*

Appendix F

Liste des Publications

F.1 Using Complex-Network Properties For Efficient Graph Analysis

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 35.92% compared to Gorder, 21.59% compared to Numbaco and 15.12% 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 35.92% compared to Gorder, 21.59% compared to Numbaco and 15.12% 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

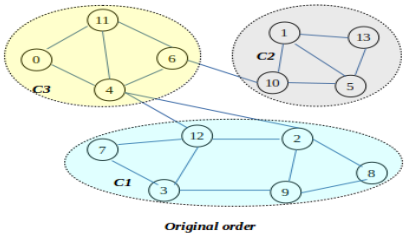


Figure 1. Running example

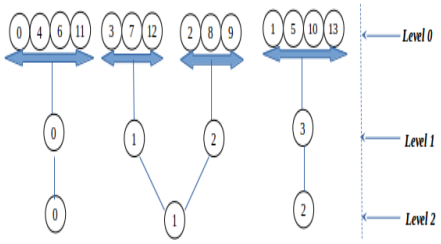


Figure 2. Running graph with Louvain

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 $S(.,.)$:

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

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: $C_{max} \leftarrow arg \max_{\forall Com_r[k]} |\{(u,v) : u \in Com_{cl}[i-1], v \in com_r[k]\}|$
5: $Com_{cl}[i] \leftarrow C_{max}$; $Com_r \leftarrow Com_r - \{C\}$; $i \leftarrow i+1$
6: **end while**
7: $i \leftarrow 1$; $offset \leftarrow 0$
8: **for all** $C \in Com_{cl}$ **do**
9: **if** $C.nb\_edge \leq Cache\_size$ **then**
10: $Com_{class}[i] \leftarrow C$,
11: $Com_{class}[i].offset \leftarrow offset$,
12: $i \leftarrow i+1$; $offset \leftarrow offset + C.nb\_node$
13: **else**
14: **for all** *subcommunity* $s\_C \in Com_{cl}$ **do**
15: $Com_{class}[i] \leftarrow s\_C$,
16: $Com_{class}[i].offset \leftarrow offset$,
17: $i \leftarrow i+1$; $offset \leftarrow offset + s\_C.nb\_node$
18: **end for**
19: **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** $i \in [1 \ldots nb_{class}]$ **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: $task[t].start \leftarrow i$, $d_{tmp} \leftarrow 0$
4: **while** $d_{tmp} < d_{max}$ **and** $i < n$ **do**
5: $d_{tmp} \leftarrow d_{tmp} + d_i$; $i \leftarrow i + 1$
6: **end while**
7: $task[t].deg \leftarrow d_{tmp}$, $task[t].end \leftarrow i + 1$; $i \leftarrow i + 1$
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	63.622 (10.52%)	48.835 (**31.32%**)	53.430 (24.85%)	47.247 (32.84%)	38.079 (**46.44%**)

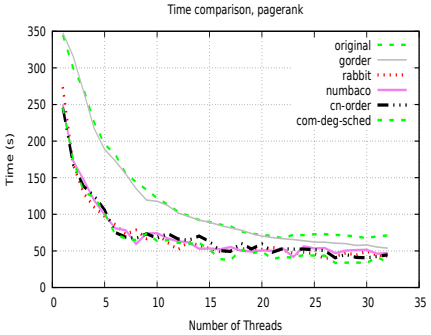


Figure 3. Time comparison with graph orders

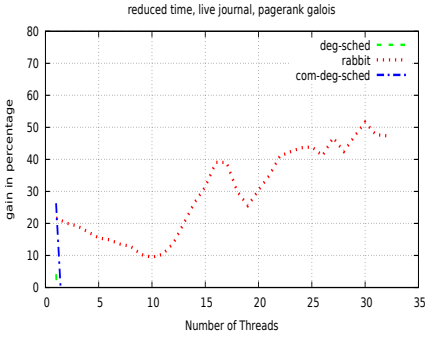


Figure 4. Performance due to scheduling & ordering

Observations done with one thread (on one core) are quite the same with multiple threads (from 2 to 32). For example with 32 threads, cn-order reduced time by 32.84%. Figure 3 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 curves are switching positions; this can be explained by the used architecture, more precisely by the L3 cache which is shared among the cores: cores benefit more and more to data loaded by others. Rabbit performance benefits more to this architecture than Numbaco, due the way it assigned numbers to nodes withing communities (see 3.2). Gorder curve, which is far compared to the last three is also switching its position with original order; this switching is also explained by the architecture.

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 15%; — compared to node-aware scheduling, using comm-deg-scheduling allows to reduce time from 32.84% (with Cn-order) to 46.44% (with Cn-order and deg-scheduling). This figure also show that it outperforms Rabbit by 15.12%.

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 35.92% compared to Gorder, 21.59% compared to Numbaco and 15.12% 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.

F.2 Social Network Ordering to Reduce Cache Misses

Numérotation des graphes sociaux

basée sur les communautés pour la réduction des défauts de cache

Thomas Messi Nguélé[1,2] — Maurice Tchuente[1] — Jean-Francois Méhaut[2]

[1] IRD, UMI 209 UMMISCO, Université de Yaoundé I, BP 337 Yaoundé Cameroun
[2] Université de Grenoble Alpes, INRIA-LIG, Corse, BP: 38400 Grenoble, France

RÉSUMÉ. L'une des propriétés des graphes sociaux est leur structure en communautés, c'est-à-dire en sous-ensembles où les nœuds ont une forte densité de liens entre eux et une faible densité de liens avec l'extérieur. Par ailleurs, la plupart des algorithmes de fouille des réseaux sociaux comportent une exploration locale du graphe sous-jacent, ce qui amène à partir d'un nœud, à faire référence aux nœuds situés dans son voisinage. L'idée de cet article est d'exploiter la structure en communautés lors du stockage des grands graphes qui surviennent dans la fouille des réseaux sociaux. L'objectif est de réduire le nombre de défauts de cache avec pour conséquence l'amélioration du temps d'exécution. Après avoir formalisé le problème de numérotation des nœuds des réseaux sociaux comme un problème d'arrangement linéaire optimal qui est connu comme NP-Complet, nous proposons *NumBaCo*, une heuristique basée sur la struture en communautés. Nous présentons pour le score de Katz et Pagerank, des simulations comparant les structures de données classiques *Bloc* et *Yale* à leurs versions exploitant *NumBaCo*. Les résultats obtenus sur une machine NUMA de 32 cœurs à partir des jeux de données amazon, dblp et web-google montrent que *NumBaCo* contribue à diminuer les défauts de cache de 62% à 80% et le temps d'exécution de 15% à 50%.

ABSTRACT. One of social graph's properties is the community structure, that is, subsets where nodes belonging to the same subset have a higher link density between themselves and a low link density with nodes belonging to external subsets. Futhermore, most social network mining algorithms comprise a local exploration of the underlying graph, which consists in referencing nodes in the neighborhood of a particular node. The idea of this paper is to use the community structure during the storage of large graphs that arise in social network mining. The goal is to reduce cache misses and consequently, execution time. After formalizing the problem of social network ordering as a problem of optimal linear arrangement which is known as NP-Complet, we propose *NumBaCo*, a heuristic based on the community structure. We present for Katz score and Pagerank, simulations that compare classic data structures *Bloc* and *Yale* to their corresponding versions that use *NumBaCo*. Results on a 32 cores NUMA machine using amazon, dblp and web-google datasets show that *NumBaCo* allows to reduce from 62% to 80% of cache misses and from 15% to 50% of execution time.

MOTS-CLÉS : Fouille de réseaux sociaux, Communauté, Défaut de cache

KEYWORDS : Social network mining, Community, Cache miss

Special issue, CRI 2015
Eric Badouel, Paulin Melatagia, Maurice Tchuenté, Editors.

ARIMA Journal, vol 24., pp. 23-47 (2017)

1. Introduction

Lorsqu'un algorithme de fouille des réseaux sociaux (comme le score de Katz [10] ou le Pagerank [15]) s'execute, il opère sur chaque nœud x du graphe en faisant le plus souvent référence aux nœuds situés dans le voisinage de x. Les structures de données utilisées dans les langages spécialisés de graphes ou les plates-formes d'analyse des graphes (Galois [14], Green-Marl [8] ou Stinger [2, 5]) ne considèrent que le voisinage direct de x.

Dans ce rapport, nous nous intéressons à la prise en compte dans la structure de données utilisée, d'un voisinage allant au delà du voisinage direct de x, ici la communauté de x. En d'autres termes, peut-on augmenter les performances des programmes d'analyse des graphes sociaux si l'on tient compte, dans la structure de données utilisée, de l'organisation en communautés des nœuds du graphe ?

La suite de cet article comprend le background à la section 2, la section 3 présente notre approche pour résoudre le problème posé, la section 4 donne l'évaluation de l'approche, la section 5 les travaux connexes, et la section 6 la conclusion et quelques perspectives.

2. Background

Avant d'entamer la section 3, nous trouvons interessant (pour faciliter la compréhension) de présenter la structure en communautés des réseaux sociaux, la gestion de la mémoire cache et la représentation des graphes.

2.1. Structure en communautés des graphes sociaux

L'une des propriétes des graphes sociaux est leur structure en communautés. Une communauté dans un graphe social est un sous-ensemble du graphe dans lequel les nœuds ont une forte densité entre eux et une faible densité avec les nœuds de l'extérieur. La détection des communautés peut-être locale ou globale. Dans la détection locale, on cherche la communauté à laquelle appartient un noeud sans forcément connaitre tout le graphe [13]. Dans la détection globale, tout le graphe est connu et on cherche à le diviser en plusieurs communautés. Par exemple, à la figure 1, un algorithme de détection de communautés divisera le graphe en deux communautés C1 = {1, 2, 5, 7} et C2 = {3, 4, 6, 8}.

Dans cet article, nous utiliserons l'algorithme de Louvain [3] pour une détection globale des communautés. Cet algorithme recherche une partition du graphe en se basant sur une fonction de qualité appelée modularité. Celle-ci attribue à une partition une valeur comprise entre -1 et 1, en fonction de la densité de liens à l'intérieur des communautés comparée à la densité des liens à l'extérieur de la communauté.

L'algorithme de Louvain commence par une partition dans laquelle chaque nœud est une communauté. Puis, l'algorithme calcule les communautés en répétant les deux phases

suivantes :

1) Pour chaque nœud i, évaluer le gain en modularité en déplaçant i de sa communauté courante vers la communauté de l'un de ses voisins. Placer i dans la communauté pour laquelle le gain est maximal (et positif).

2) Générer un nouveau graphe dans lequel les nœuds sont les communautés détectées à l'étape précédente. Revenir à la première étape avec en entrée ce nouveau graphe.

Ces deux phases sont répétées jusqu'à ce que le gain en modularité ne soit plus possible.

2.1.1. Structure imbriquée de Louvain

L'algorithme de Louvain retourne une structure hierarchique (ou imbriquée) qui est telle que les communautés obtenues à la dernier étape sont constituées des sous-communautés obtenues aux étapes précédentes. Cette structure sera utilisée dans la suite pour optimiser le stockage des nœuds.

2.2. Gestion de la mémoire cache

Un processeur qui veut accéder à une donnée pendant l'exécution d'un programme recherche d'abord l'entrée correspondante dans le cache. Si la donnée n'est pas présente, il y a défaut de cache.

Il existe trois principales catégories de défaut de cache qui sont :

– les défauts de cache obligatoires causés par la première référence à une donnée,

– les défauts de cache conflictuels causés par les données ayant la même adresse dans le cache,

– les défauts de cache capacitifs causés par le fait que les données d'un programme ne peuvent pas suffire dans le cache. C'est cette catégorie que nous considérons dans cet article.

Lorsque survient un défaut de cache, l'un des algorithmes classiques suivant est exécuté pour ramener la donnée de la mémoire dans le cache :

– algorithme optimal (la ligne de cache qui ne sera pas utilisée pour la plus grande période de temps est remplacée),

– algorithme aléatoire, – LRU *Least Recently Used*,

– FIFO *First In First Out*, – LFU *Least Frequently Used*, ...

Étant donné que les processeurs généralistes (Intel, AMD, ARM) implémentent directement l'un de ces algorithmes dans leur matériel, pour bénéficier de l'efficacité de la mémoire cache, l'utilisateur devrait s'assurer que les structures de données qu'il utilise (graphe dans notre cas) sont bien organisées pendant l'exécution du programme.

2.3. Réprésentation des graphes

Soit n le nombre de nœuds et m le nombre d'arêtes. Considérons le graphe $G1$ pondéré et non orienté de la figure 1. $G1$ a n=8 et m=11X2 (une arête est comptée double car le graphe est non-orienté).

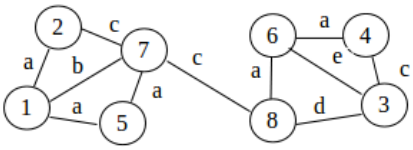


Figure 1 – Graphe $G1$

2.3.1. Représentation avec une matrice d'adjacence, $espace\ O(n^2)$

Une façon simple de représenter ce graphe est d'utiliser la représentation matricielle. Dans ce cas, on se sert d'une matrice d'adjacence M où $M(i,j) = m_{ij}$ est le poids de l'arête entre les nœuds i et j. Mais celle-ci n'est pas très appropriée pour les graphes sociaux car les matrices résultantes sont souvent creuses. Par exemple, pour le graphe de la figure 1, on utilise 8X8 = 64 cases alors que 42 cases (66%) sont gaspillées pour stocker les zéros (voir tableau 1).

	1	2	3	4	5	6	7	8
1		a			a		b	
2	a						c	
3				c		e		d
4			c			a		
5	a						a	
6			e	a				a
7	b	c			a			c
8			d			a	c	

Tableau 1 – Matrice représentant le graphe (G)

2.3.2. Représentation sous forme de Yale, $espace\ O(n + 2m)$

La représentation souvent adoptée par certains langages spécialisés de graphe (à l'instard de Galois [14] et Green-Marl [8]) est celle de Yale [6]. Cette représentation se sert de trois vecteurs permettant de simuler la matrice d'adjacence (M pour notre exemple) :

– Un vecteur A utilisé pour stocker le poids de chacune des arêtes. Dans le tableau 2, le vecteur A est utilisé pour stocker les poids des 22 arêtes du graphe G1.

– Un autre vecteur JA donnant l'extrémité j de chacune des arêtes dont le poids est stocker dans A. De ce fait, A et JA ont la même taille (22 pour le graphe G1).

– Et un dernier vecteur IA qui donne l'indice dans A du premier élément non nul de chaque ligne de la matrice simulée (M). La dernière case contient le nombre d'arête + 1.

Par exemple, le premier élément non nul de la première ligne de M est stocké à case 1 de A ; le premier élément non nul de la deuxième ligne de M est stocké à la case 4 de A ; le premier élément non nul de la troisième ligne de M est stocké à la case 6 de A ; ... ; le premier élément non nul de la huitième ligne de M est stocké à la case 20 de A et la dernière case contient le nombre d'arêtes 22 + 1 = 23.

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

Tableau 2 – Répresentation de Yale du graphe $G1$

2.3.3. Représentation avec des listes d'adjacence, $espace\ O(n + 2m)$

D'autres représentations peuvent être utilisées : le graphe est alors représenté par un vecteur de nœuds, chaque nœud pouvant être relié

1) à un bloc de ses voisins, la taille du bloc étant variable [17] (voir figure 2 a)-)

2) à une liste chainée de blocs (de taille fixe) de ses voisins, adaptée aux graphes dynamiques et utilisée par la plateforme Stinger [2,5] (voir figure 2 b)-).

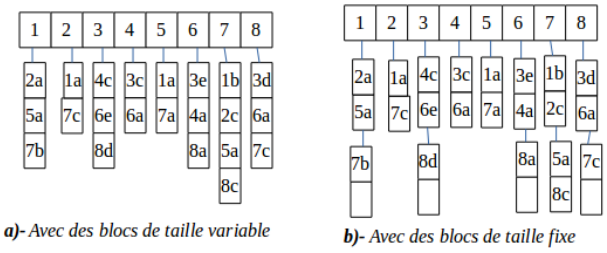


Figure 2 – Adjacency list representations of $G1$

Aucune de ces représentations ne tire profit du "regroupement en communautés" des nœuds du graphe pour réduire le temps d'exécution des algorithmes des réseaux sociaux, car ce n'était pas leur but.

3. Numérotation des graphes sociaux

3.1. Idée

L'idée est de faire en sorte que, chaque fois qu'un nœud est en cours de traitement, les autres nœuds membres de sa communauté se retrouvent dans le même cache mémoire que ce nœud. Ainsi, les données correspondant à une communauté doivent être consécutives en mémoire.

3.1.1. Structure en communautés du graphe

Le graphe est maintenant perçu comme un ensemble de communautés (détectées par l'algorithme de Louvain [3]). On procède ainsi à une renumérotation du graphe en suivant la structure imbriquée de Louvain (voir section 2.1.1) : les nœuds appartenant à une même communauté ou à une même sous-communauté ont des numéros consécutifs. Ceci aura pour effet de les rapprocher en mémoire pendant l'exécution d'un programme d'analyse des réseaux sociaux.

À la figure 3, le graphe comporte 3 communautés : C1, C2 et C3. À la partie droite, les nœuds ont des numéros consécutifs dans chaque communauté et dans chaque sous-communauté. Par exemple C3 est subdivisée en deux sous-communautés ; et les nœuds de chaque sous-communautés sont consécutifs.

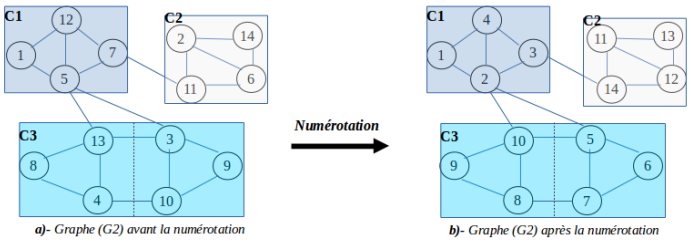


Figure 3 – Représentation de (G2) tenant compte des communautés

3.1.2. stockage du graphe

La structure en communautés est incorporée dans les modes de répresentation du graphe (rappelés à la section 2.3). Dans la nouvelle représentation obtenue, les voisins sont stockés de sorte que, lors d'un traitement, la priorité est accordée aux nœuds situés dans une même sous-communauté ou dans une même communauté.

La figure 4 donne une représentation par liste d'adjacence du graphe (G2) avant et après la prise en compte de la structure en communautés. On peut remarquer qu'après cette prise en compte, les nœuds appartenant à la même communauté sont maintenant plus groupés (distinction avec les couleurs).

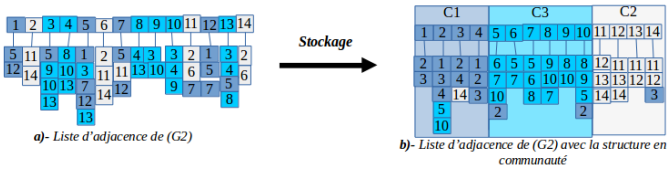


Figure 4 – Liste d'adjacence de (G2) tenant compte des communautés

Concernant le stockage des voisins, en considerant le nœud 10 (après numérotation), on remarque que ses voisins sont stockés dans cet ordre 8, 9, 5 et 2 : 8 et 9 sont d'abord stockés parce qu'ils sont dans la même sous-communauté que 10 (voir figure 3) ; ensuite on stocke 5 parce qu'il est dans la même communauté que 10 ; 2 est stocké en dernier parce qu'il n'est pas dans la même communauté que 10.

L'astuce utilisée pour rapprocher les nœuds en mémoire est la renumérotation du graphe en tenant compte des communautés. Dans la suite, nous la présenterons en détail après avoir montré qu'elle est une heuristique d'un problème plus général, *le problème de numérotation des graphes sociaux pour la réduction des défauts da cache*.

3.2. Formalisation du problème

3.2.1. Modélisation des défauts de cache

Nous considérons uniquement les défauts de cache capacitifs causés par le fait que les données utilisées lors de l'exécution d'un programme ne peuvent pas tenir en totalité dans le cache mémoire. On considère ici un cache de données avec une seule ligne encore appelée bloc, et une mémoire principale dont la taille est de t blocs (cf figure 5).

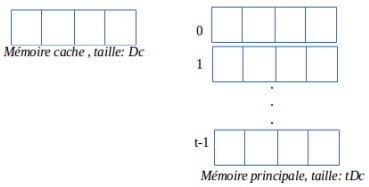


Figure 5 – Une ligne dans le cache et t blocks dans la mémoire principale

Nous notons :

– D_c : la taille de la mémoire cache,

– N : l'ensemble des nœuds

– π : une permutation dans l'ensemble des nœuds

– b : la fonction qui donne le numéro du bloc auquel appartient un nœud x, $b(x) = x$ **Div** D_c (**Div** étant la division entière)

Lorsqu'un processeur traite un nœud x (qui se trouve donc dans le cache) et essaye d'accéder à un autre nœud y, deux situations sont possibles :

– si les deux nœuds sont dans le même bloc mémoire $b(x) = b(y)$, alors y se trouve aussi dans le cache

– si x et y ne sont pas dans le même bloc mémoire, on dit qu'il y a un défaut de cache.

Le défaut de cache peut donc être modélisé par la fonction σ définie par :

$$\begin{array}{rccl} \sigma : & N \times N & \longrightarrow & \{0,1\} \\ & (x,y) & \longmapsto & \sigma(x,y) = \begin{cases} 0 & si\ b(x) - b(y) = 0 \\ 1 & sinon \end{cases} \end{array}$$

3.2.2. Complexité du problème

Supposons qu'un programme P dans son exécution sur un graphe social $G = (N, E)$, fasse référence à une séquence ordonnée de nœuds $S_k = (n_1, n_2, n_3, ..., n_k)$. Le nombre de défauts de caches provoqué par cette exécution est donné par :

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

On aimerait alors trouver une numérotation (permutation) π des nœuds de ce graphe pour que P produise le minimum de défauts de cache (min_{dc}). Autrement dit,

$$Soit\ G = (N, E), min_{dc} \in \mathbb{N}, \begin{cases} - & on\ cherche & \pi : & N & \longrightarrow & N \\ & & & n & \longmapsto & \pi(n) \\ - & pour\ que & \sum_i^k \sigma(\pi(n_i), \pi(n_{i-1})) \leq min_{dc}, & & & \\ & avec\ N_k = (n_1, ..., n_k)\ et\ n_i \in N. & & & & \end{cases}$$

Proposition 3.1 (relatif à la complexité du problème) *Le problème de numérotation des graphes sociaux (PNGS) tel que défini plus haut est NP-complet.*

Démonstration :*Nous nous servons du problème d'arrangement linéaire optimal (PALO) connu comme étant NP-complet [7]. En rappel, ce problème est défini comme suit :*

$$Soit\ G = (N, A), min \in \mathbb{N} \begin{cases} - & on\ cherche & \pi : & N & \longrightarrow & N \\ & & & n & \longmapsto & \pi(n) \\ - & pour\ que & \sum_{\{n_i, n_j\} \in A} |\pi(n_i) - \pi(n_j)| \leq min & & & \end{cases}$$

Dans cette démonstration, il nous faut montrer que toute instance de PALO peut être transformée en temps polynomial en une instance de PNGS. A cet effet, il suffit de remar-

quer que toute instance de PALO est une instance de PNGS en considérant l'exécution de la boucle qui balaie toutes les arêtes de G.

3.2.3. Heuristique basée sur la struture en communautés

Le problème de numérotation des graphes sociaux (PNGS) peut être résolu avec une heuristique basée sur la structure en communautés des graphes sociaux. L'algorithme 1 (NumBaCo) est une heuristique pour PNGS.

Algorithm 1 : **Num**érotation **Ba**sée sur les **Co**mmunautés (**NumBaCo**)

Entrée : $G = (N, E)$, un graphe social
Sortie : $G' = \pi(G)$, π est un ordonnancement (permutation), le voisinage de chaque nœud x' de G' est stocké en respectant la structure imbriquée de Louvain

1: $Com \leftarrow detect\_comm\_Louvain(G)$
2: $Com_{cl} \leftarrow comm\_des\_comm(Com)$
3: $\pi \leftarrow numerotation\_graphe(Com_{cl}, G)$
4: $G_1 \leftarrow stockage\_des\_voisins(Com, G)$
5: $G' \leftarrow nouveau\_graph(G_1, \pi)$
6: **return** G'

3.2.3.1. $Com \leftarrow detect\_comm\_Louvain(G)$

Il s'agit de l'algorithme de Louvain tel que décrit plus haut (voir section 2.1). Il prend en paramètre un graphe (G) et retourne l'ensemble de communautés de (G) sous forme hierarchique ou imbriquée (voir section 2.1.1). Cet algorithme s'exécute en $O(nlogn)$, n étant le nombre de nœuds de (G). Une exécution de cet algorithme pourra donner la configuration à la figure 3a) où le graphe (G2) est divisé en 3 communautés C1, C2 et C3.

3.2.3.2. $Com_{cl} \leftarrow comm\_des\_comm(Com)$

Cet algorithme permet de classer les communautés en fonction de leur affinité (nombre d'arêtes qu'elles partagent entre elles). Il fonctionne comme l'algorithme de Louvain :

– La communauté i est placée dans la communauté voisine avec laquelle elle partage le plus grand nombre d'arêtes.

– On repète l'opération précedente jusqu'à ce qu'il ne reste qu'une seule communauté.

– Le classement des communautés initiales est donné par leur ordre d'inclusion dans la communauté finale Com_{cl}. L'opération la plus couteuse ici le calcul des affinités.

L'algorithme s'exécute alors en $O(nk + C^2)$, où k est le dégré moyen des nœuds du graphe et C le nombre de communautés contenues dans Com.

3.2.3.3. $\pi \leftarrow numerotation\_graphe(Com_{cl}, G)$

Cette fonction est utilisée pour générer une nouvelle numérotation du graphe :

– Les nœuds appartenant à la même communauté ou sous-communauté ont des numéros consécutifs

– Com_cl est utilisé pour décider quelle communauté (ou sous-communauté) vient avant l'autre en mémoire ; les numéros des nœuds des communautés qui viennent en premier ont des numéros plus petits.

Cette étape s'exécute en $O(n)$.

3.2.3.4. $G_1 \leftarrow stockage\_des\_voisins(Com, G)$

Cette fonction permet de changer l'ordre de stockage des voisins.

– Le voisinage de chacun nœud suit désormais la structure imbriquée de Louvain (voir section 2.1.1).

– Par exemple, le nœud 10 de la figure 3b) a l'ordre de stockage de ses voisins modifié à la figure 4b). Dans un stockage classique, on respecte plutôt l'ordre chronologique des nœuds.

La complexité de cette étape est de $O(nkC)$.

3.2.3.5. $G' \leftarrow nouveau\_graph(G_1, \pi)$

Ici, le nouveau graphe est généré avec une nouvelle numérotation et avec un stockage des voisins qui respecte la structure imbriquée de Louvain. La complexité ici est de $O(nk)$.

Ainsi la complexité totale de l'algorithme est de $O(nlogn + (nk + C^2) + n + nkC + nk) = O(nlogn + n(K(2 + C) + 1) + C^2)$.

3.2.4. Quelques propriétés : gain dû à la numérotation

Étant donné un programme P d'analyse des réseaux sociaux, le gain désigne ici la réduction du nombre de défauts de cache provoquée par l'exécution de P lorsque les nœuds ont été stockés en mémoire en suivant l'algorithme de numérotation présenté plus haut.

Propriéte 3.1 (Exploration des nœuds) *Le gain obtenu par cette numérotation depend de l'exploration des noeuds du graphe pendant l'exécution du programme :*

1) Il est grand pour une exploration locale du graphe (à partir d'un nœud, on fait réference aux nœuds situés dans son voisinage).

2) Il est petit pour une exploration non locale du graphe (à partir d'un nœud, on fait réference aux nœuds situés en dehors de son voisinage).

Indications : *Dans le premier cas, comme les nœuds successifs sont plus rapprochés en mémoire, on a moins de défauts de cache ; le gain est donc grand.*

Dans le deuxième cas, les nœuds successifs étant éloignés en mémoire, il y a plus de défauts de cache ; le gain est donc plus petit.

Propriéte 3.2 (Rangement initial des nœuds) *Le gain obtenu par cette numérotation depend du rangement initial (avant l'usage de NumBaCo) des noeuds dans le graphe :*

1) Il est grand lorsque les noeuds inter-communautaires (de chacune des communautés) ont des numéros non consécutifs et très éloignés.

2) Il est petit lorsque les noeuds inter-communautaires (de chacune des communautés) ont des numéros consécutifs (gain nul) ou proche (gain petit).

Indications : *Dans le premier cas, le rapprochement des nœuds voisins est effectif. Ainsi dans une exploration locale du graphe, il y aura moins de défauts de cache avec la numérotation générée par l'algorithme 1.*

Dans le deuxième cas, la numérotation générée par l'algorithme 1 sera presqu'identique à la numérotation initiale. Ainsi le gain sera faible.

4. Évaluation

Les expérimentations ont été menées sur la machine NUMA32, 4 nœuds (*numa node*) de 8 cœurs chacun, soit au total 32 cœurs pour 64 Go. Chaque noeud est de type Intel Xeon avec les caractéristiques 2.27GHz, L1 32KB, L2 256KB, L3 24MB, pas d'*Hyper-Threading*. La figure 6 présente un *numa node*.

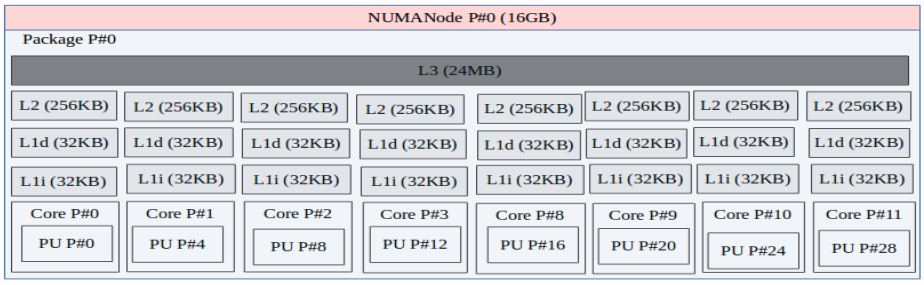


Figure 6 – *Numa node* : 8 cœurs, L1 et L2 privé, L3 partagé

Pour cette évaluation, nous avons utilisé deux applications d'analyse des réseaux sociaux : le score de Katz [10] et le Pagerank [15]. Chacune de ces applications a été implémentée avec deux structures de données pour la repésentation des graphes :

– par liste d'adjacence, chaque nœud étant relié à un bloc (de taille variable) de ses voisins : on l'appellera *bloc*

– sous forme de Yale : on l'appellera *yale*

Chacune de ces deux structures a ensuite été réorganisée en utilisant l'algorithme NumBaCo présentée à la section 3.2.3. Nous notons chacune des structures résultantes par *b_numbaco* et *y_numbaco*.

4.1. Score de katz

Le score de Katz [10] est utilisé comme une mesure de similarité basée sur les distances entre les nœuds. Le score de Katz entre deux nœuds x et y est donné par la formule :

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

Où :

– L représente la taille maximale d'un chemin.

– $paths_{x,y}^{<l>}$ est l'ensemble des chemins de longueur l entre x et y, et $|paths_{x,y}^{<l>}|$ représente leur nombre.

– $0 < \beta < 1$. β est choisi tel que les chemins avec un l grand contribuent moins à la somme que les chemins avec un l petit.

Nous avons réalisé une implémentation multithreadée de l'algorihme de katz. Les nœuds sont rangés dans une liste chainée qui est ensuite parcourue en parallèle. Pour chaque nœud, la fonction **computeKatzNode** est invoquée (voir algorithme 2). L est responsable de la différence de temps d'exécution pour un même graphe.

Algorithm 2 Katz multi-threadé

```
1:  Global nodeList, G, β, L, K_sc
2:  do_work()
3:  while nodeList ≠ ∅ do
4:      x ← atomic_dequeue(nodeList)
5:      K_sc[x] ← computeKatzNode(x, G, β, L)
6:  end while
7:
8:  main()
9:  nodeList ← generate_nodeList(G)
10: for i = 1 to n_threads do
11:     spawn_thread(do_work())
12: end for
13: wait_every_child_thread()
14: output(K_sc)
```

Pour tout nœud x de G, on peut démontrer que les nombres de chemins à partir de ce nœud vers les autres nœuds se calculent ainsi qu'il suit (N_i, et L_i représentent respectivement les voisins et les nombres de chemins d'ordre i) :

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

Ceci nous permet d'établir l'algorithme 3. La clé réside dans le calcul du vecteur de nombre de chemins $cLenPath$ à la ligne 6 avec la fonction **updateLenPath**() développée entre les lignes 15 et 26). À chaque valeur de l, avant de passer à la valeur suivante, le score de katz est mis à jour (lignes 7 à 10) ; l'ensemble des nœuds courants et le tableau des nombres de chemins sont également mis à jour (lignes 11 et 12).

Algorithm 3 Score de Katz entre x et tout nœud atteignable avec L

```
1:  computeKatzNode(x, G, β, L)
2:  dNeig ← G.neighbors(x)
3:  pNeig ← dNeig
4:  pLenPath[{dNeig}] ← 1
5:  for l = 2 → L do
6:      [cNeig, cLenPath] ← updateLenPath(pNeig[ ], pLenPath[ ])
7:      for all (t ∈ cNeig) and (t ∉ dNeig) do
8:          katz[t] ← katz[t] + β^l cLenPath[t]
9:          accessibleNeig.add(t)
10:     end for
11:     [pNeig, cNeigh] ← [cNeig, empty()]
12:     [pLenPath, cLenPath] ← [cLenPath, empty()]
13: end for
14: return buildLign(x, katz[ ], accessibleNeig[ ])
15: updateLenPath(pNeig[ ], pLenPath[ ])
16: for all y ∈ pNeig do
17:     for all z ∈ G.neighbors(y) do
18:         if z ∈ cNeig then
19:             cLenPath[z] ← cLenPath[z] + pLenPath[z]
20:         else
21:             cLenPath[z] ← pLenPath[z]
22:             cNeig.add(z)
23:         end if
24:     end for
25: end for
26: return [cNeig, cLenPath]
```

4.2. Pagerank

Pagerank [15] est un algorithme utilisé par Google pour classifier les pages dans le web. Le Pagerank d'une page x est donné par la formule suivante :

$$PR(x) = (1-d) + d \sum_{y \in N_{in}(x)} \frac{PR(y)}{|N_{out}(y)|} \tag{3}$$

Où :

– d est la probabilité de suivre cette page, $(1-d)$ la probabilité de suivre une autre page.

– $N_{in}(x)$ est l'ensemble des voisins entrant de x.

– $N_{out}(y)$ est l'ensemble des voisins sortant de y.

Nous avons utilisé une implémentation *posix thread* proposée par *Nikos Katirtzis*[1]. Cette implémentation utilise la structure de données *bloc*.

4.3. Résultats

Pour cette évaluation, nous nous sommes servis des graphes amazon, dblp et web-google [19].

4.3.1. Résultats sur amazon

Les nœuds représentent les produits vendus en ligne par le site amazon. Il existe un lien entre deux produits si ceux-ci sont fréquemment achetés ensemble. Les communautés (qui seront détectées ici par l'algorithme de Louvain) peuvent être considérées comme des ensembles de produits appartenant à la même catégorie. Le graphe considéré ici (après suppression des nœuds isolés) a 269906 nœuds et 1851744 arêtes.

4.3.1.1. Diminution du temps d'exécution

La figure 7 présente les temps d'exécution obtenus sur la machine décrite plus haut. En haut, nous avons la comparaison avec la structure *bloc* et en bas la comparaison avec la structure *yale*. Nous lisons chaque partie de la gauche vers la droite et de haut en bas. Considérons la première partie (*a-comparaison avec bloc*). Au premier cadrant, la courbe *b_numbaco* reste en dessous de la courbe *bloc*. Ceci traduit bien le fait que l'algorithme *NumBaco* contribue à réduire le temps d'exécution. Dans le troisième cadrant, on peut voir les différences de temps (en seconde) entre les deux structures. Ces différences se réduisent avec le nombre de cœurs (même variation que les temps d'exécution). Elles sont plus significatives en observant la courbe du quatrième cadrant qui présente les pourcentages des temps reduits en fonction du nombre de cœurs.

Cette courbe montre que les performances augmentent en fonction du nombre de cœurs, variant de 6.5% (avec 1 cœur) à 21.6% (avec 32 cœurs). Ceci peut s'expliquer par l'architecture utilisée (voir figure 6), plus précisement par le cache L3 qui est partagé entre les cœurs : les cœurs profitent de plus en plus des données chargées par les autres. Cette explication justifie aussi le speedup obtenu (deuxième quadrant, 32 cœurs) qui est de 27.8 avec *NumBaco* comparé à 23.3 sans *NumBaco*.

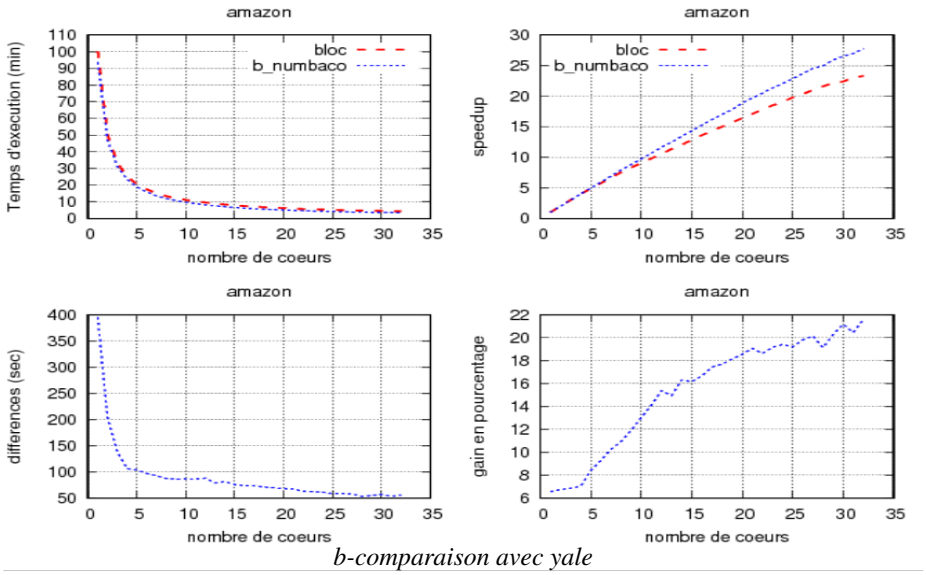
Nous faisons des observations similaires dans la deuxième partie (*b-comparaison avec yale*). Cette fois ci, *NumBaco* permet de réduire le temps d'exécution jusqu'à 20.6%. Et le speedup est de 27.7 avec *NumBaco* comparé à 23.4 sans *NumBaco*.

4.3.1.2. Diminution des défauts de cache

Pour vérifier que les temps d'exécution observés étaient liés au nombre de défauts de caches causés par le programme, nous avons lancé le programme avec l'outil *perf* [2]. La

1. https://github.com/nikos912000/parallel-pagerank
2. https://perf.wiki.kernel.org/index.php/Tutorial

a-comparaison avec bloc



b-comparaison avec yale

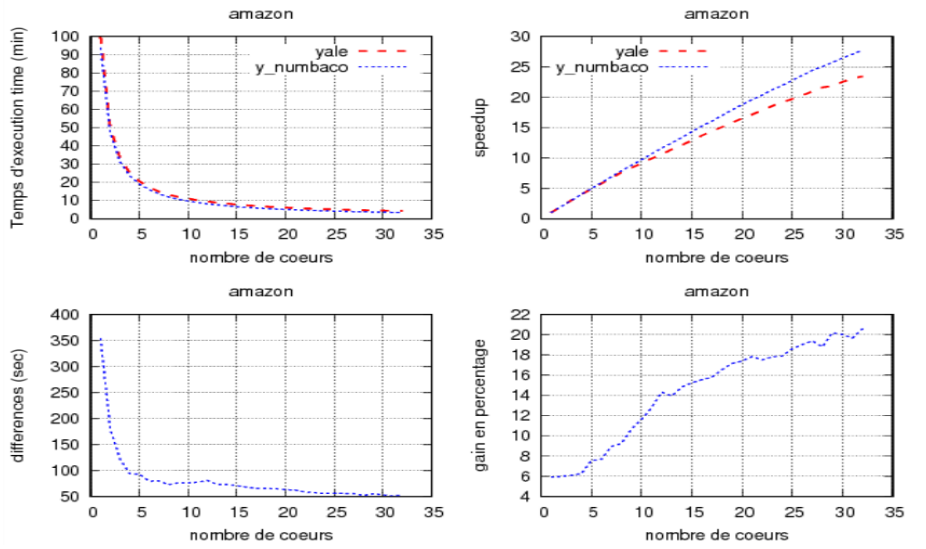
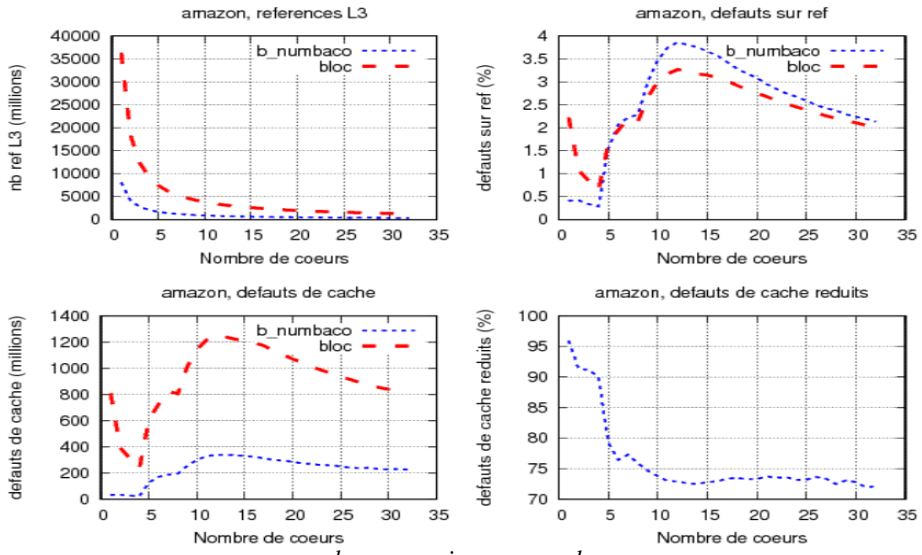


Figure 7 – Diminution du temps d'exécution – amazon, Katz

a-comparaison avec bloc



b-comparaison avec yale

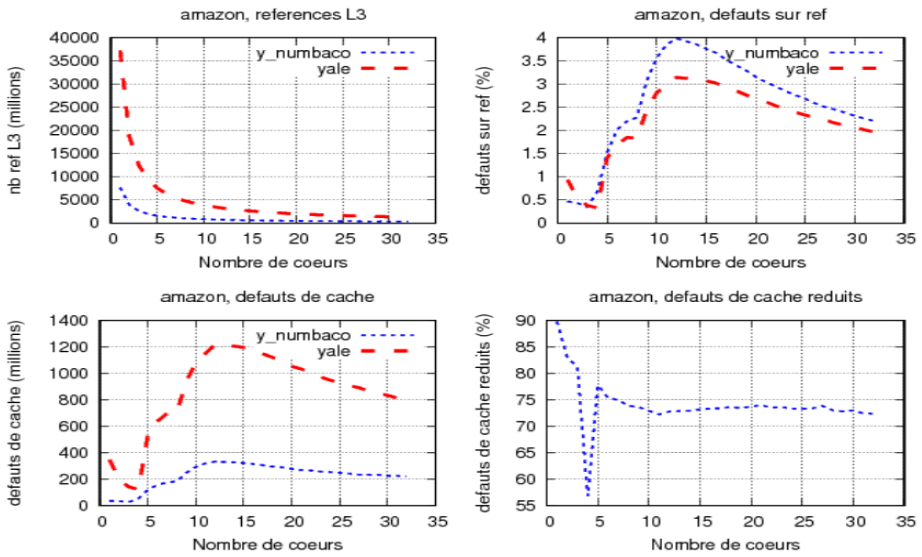


Figure 8 – Diminution des défauts de cache – amazon, Katz

figure 8 présente les résutats obtenus pour les évenements "cache-references" et "cache-misses".

Dans chaque partie, le premier cadrant correspond aux courbes des évènements "cache-references", le troisième cadrant correspond aux courbes de l'évènements "cache-misses". Le deuxième cadrant correspond au rapport (en %) entre les évènements "cache-misses" et "cache-references". Le quatrième cadrant donne le pourcentage des nombres de défauts de caches réduits l'usage de *NumBaCo*.

Considérons la première partie (*a-comparaison avec bloc*). Dans le premier cadrant, la courbe *b_numbaco* reste en dessous de la courbe *bloc*. Ceci signifie que *NumBaCo* permet de réduire le nombre de références au cache L3. En effet, les données (les nœuds) étant mieux organisées (avec la structure *b_numbaco*), les caches L2 et L1 se retrouvent plus sollicités. Ceci contribue à moins referencer le cache l3.

L'effet direct de la réduction du nombre de références au cache L3 est la diminution du nombre de défauts de cache observable au troisième cadrant. Ici, on observe bien que la courbe *bloc* reste au dessus de la courbe *b_numbaco* ; ce qui montre qu'en prenant en compte la structure en communautés, on réduit le nombre de défauts de cache. Le quatrième cadrant nous permet de voir qu'on réduit les defauts de caches de 95% (1 cœur) à 73% (32 cœurs).

Des observations similaires sont effectuées dans la deuxième partie (*b-comparaison avec yale*).

4.3.2. Résultats sur dblp

Les nœuds correspondent aux auteurs des articles scientifiques en informatique. Il existe un lien entre deux auteurs s'ils ont été co-auteurs d'au moins un article. Les communautés (qui seront détectées ici par l'algorithme de Louvain) correspondent aux auteurs qui ont publié dans un même journal ou dans une même conférence. Le graphe considéré ici (après suppression des nœuds isolés) a 195310 nœuds et 2099732 arêtes.

4.3.2.1. Diminution du temps d'exécution

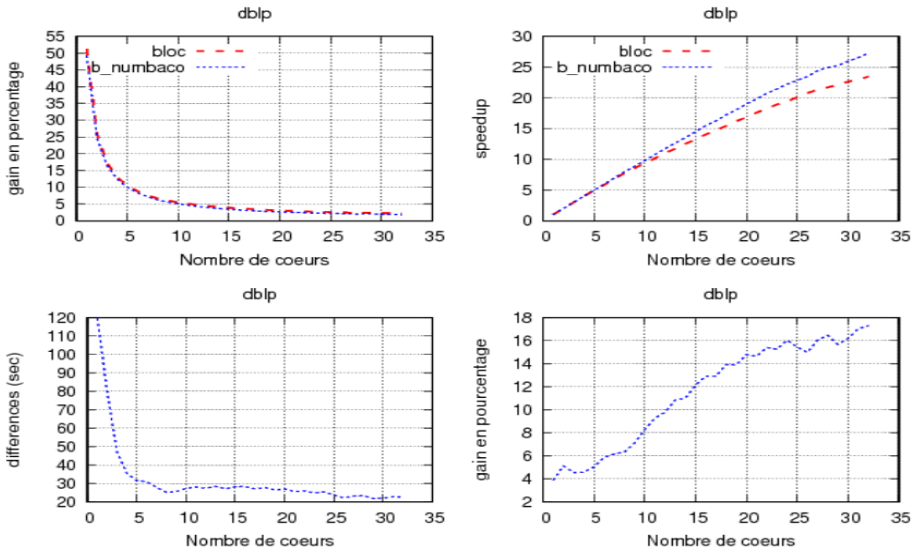
La figure 9 présente les résultats les temps d'exécution obtenus.

Les courbes ont les mêmes allures que dans le cas précédent (cas d'amazon). La différence se fait au niveau des performances observées. Avec le jeu de données dblp, l'usage de *NumBaco* permet de réduire le temps d'exécution jusqu'à 17% par rapport à *bloc* et 15% par rapport à *yale*. Dans ce cas, le speedup à 32 cœurs est de 27.3 (comparé à 23.5 avec *bloc*) et 26.7 (comparé à 23.5 avec *yale*).

4.3.2.2. Diminution des défauts de cache

Comme dans le cas du jeu de données amazon et comme le montre la figure 10, la diminution des temps d'exécution est aussi liée à la diminution des défauts de cache. Ici à 32 cœurs, les defauts de cache sont réduits de 64% (comparé à *bloc*) et 62% (comparé à *yale*).

a-comparaison avec bloc



b-comparaison avec yale

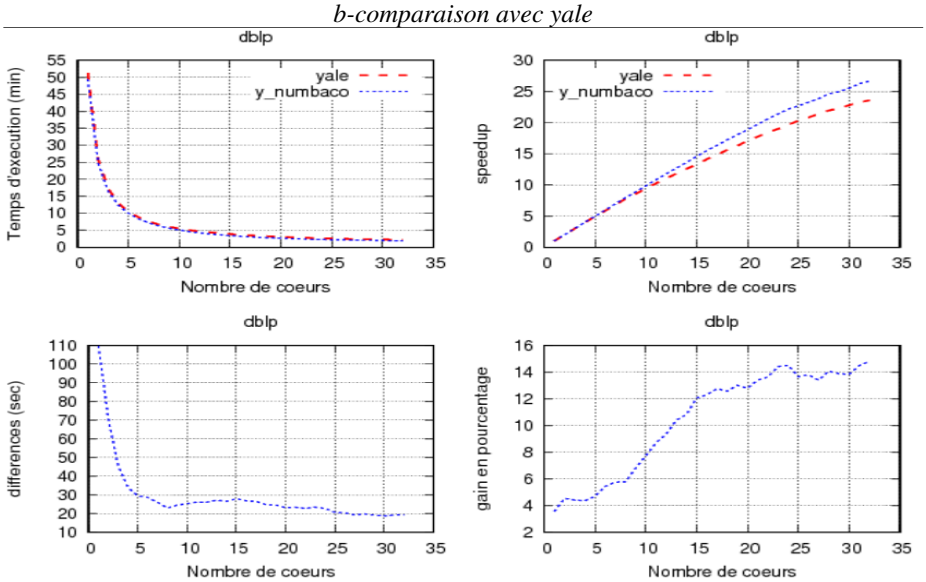
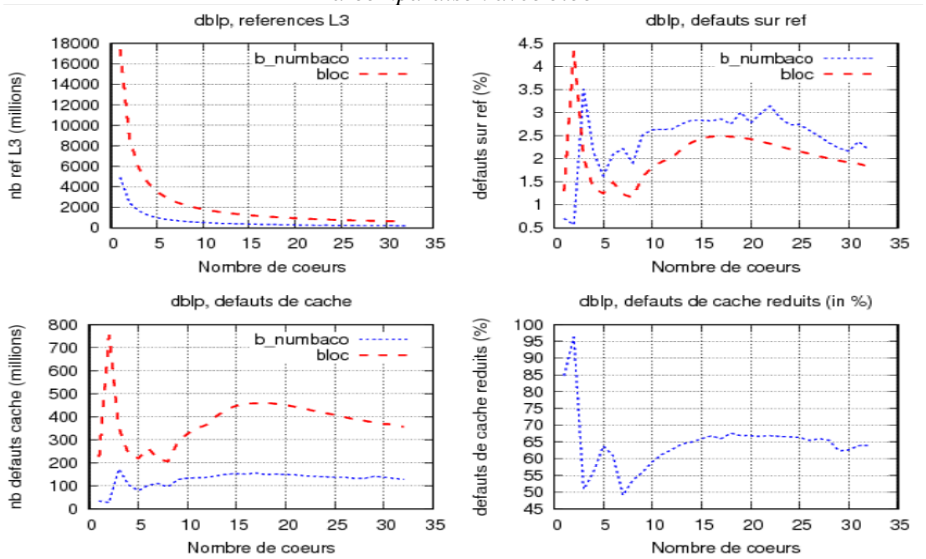


Figure 9 – Diminution du temps d'exécution – dblp, Katz

a-comparaison avec bloc



b-comparaison avec yale

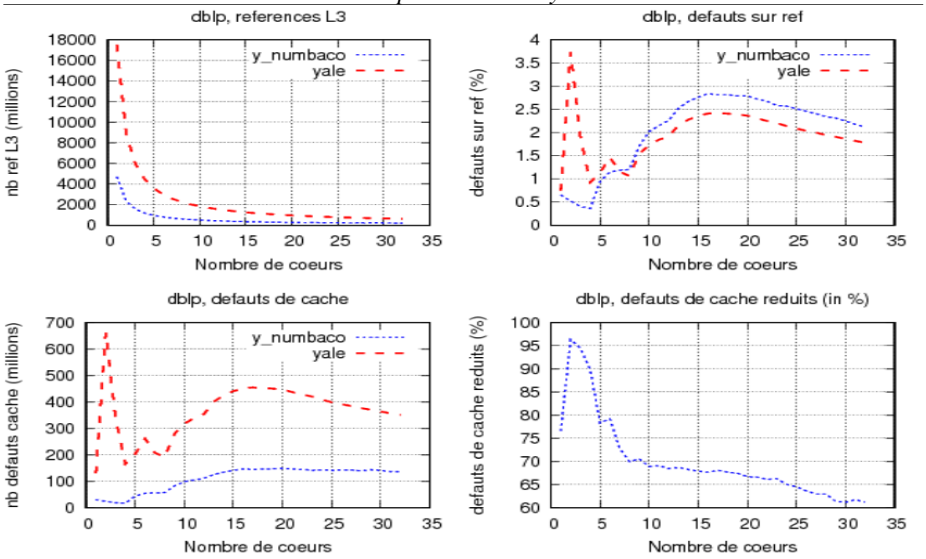


Figure 10 – Diminution des défauts de cache – dblp, Katz

4.4. Résultats sur Web-goole avec Pagerank

La figure 11 montre que NumBaCo permet de réduire plus de 50% du temps d'exécute. Cette réduction du temps d'exécution est due à la reduction de près de 80% des défauts de cache que l'on peut observer à la figure 12.

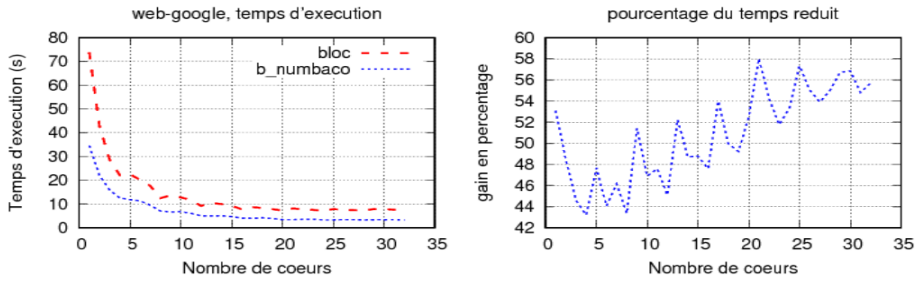


Figure 11 – Réduction du temps d'exécution – web-google, Pagerank

4.5. Discussion

4.5.1. Différence de performance

La différence de performance observée entre les jeux de données amazon et dblp s'explique par le rangement initial des nœuds (Propriété 3.2) : les nœuds inter-communautaires du graphe initial d'amazon ont des numéros plus éloignés que les numéros des nœuds inter-communautaires du graphe initial de dblp. (Pour le vérifier, nous avons calculé la différence de coût $\sum_{(n_i,n_j)} |\pi(n_i) - \pi(n_j)|$ avant et après usage de l'algorithme de numérotation, cette différence était plus élévée avec le graphe amazon ; ce qui traduit un meilleur gain).

La différence de perfomance entre le Pagerank (80% des défauts de cache pour 50% du temps d'exécution) et le score de Katz (73% des défauts de cache pour 21% du temps d'exécution) s'explique l'exploration locale (Propriéte 3.1) : dans le cas du Pagerank, les nœuds successifs auxquels le programme fait reférence sont plus rapprochés en mémoire (ce qui cause moins de défauts de cache) ; et dans le cas du score de Katz, les noeuds successifs sont moins rapprochés (ce qui cause plus de défauts de cache).

4.5.2. Surcoût induit par NumBaCo

L'algorithme de numérotation présenté plus haut a pour vocation d'être exécuté avant un programme d'analyse des réseaux sociaux. Ceci afin d'avoir une organisation des nœuds en mémoire permettant de réduire au maximum les défauts de cache. Les performances du programme sont alors améliorées. Toutefois, pour atteindre cet objectif, il faudra que le programme en question ait une complexité plus élévée que l'algorithme de numérotation. Dans les expérimentations que nous proposons dans ce rapport, l'al-

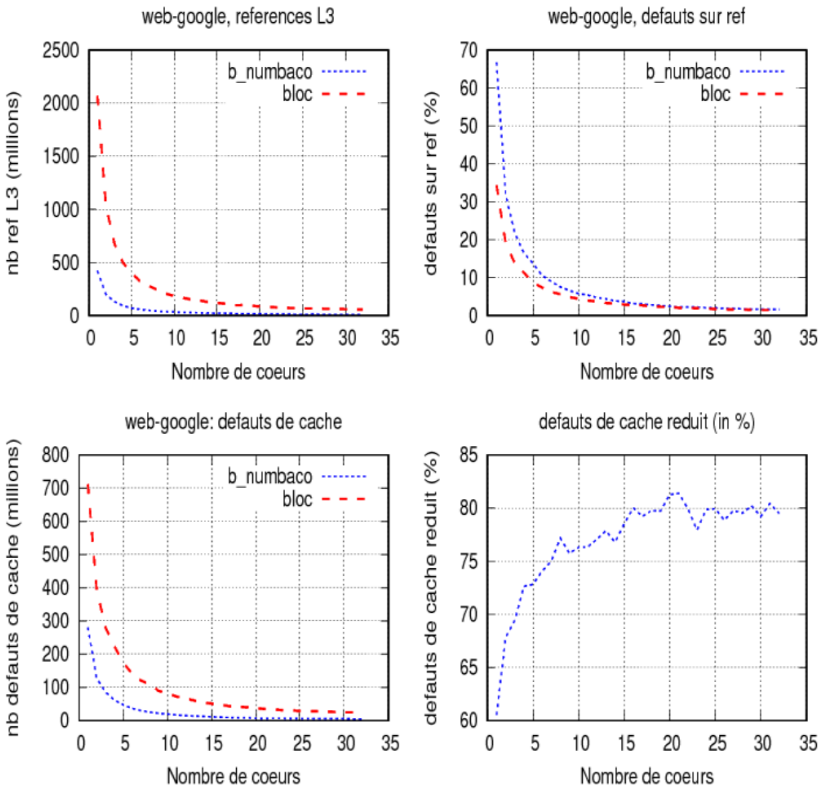


Figure 12 – Réduction des défauts de cache – web-google, Pagerank

gorithme de katz a une plus grande complexité ($O(n^3)$ dans le pire des cas, comparé à $O(nlogn)$ de l'algorithme de numérotation).

Plus concrètement, en relevant le temps mis par *NumBaCo*, nous avons obtenu *1675 ms* pour amazon et *1683 ms* pour dblp (voir tableau 3). Ce qui est relativement petit par rapport au gain obtenu. Par exemple avec le jeu de données amazon (voir tableau 4), en comparant avec la structure *bloc*, on a obtenu le gain *389276 ms* sur 1 cœur, *72668 ms* sur 19 cœurs et *54252 ms* sur 32 coeurs (ce qui correspondent à environ *232 fois*, *43 fois* et *32 fois* le temps perdu).

Jeu de données	amazon	dblp
Temps (*NumBaCo*)	1675 ms	1683 ms

Tableau 3 – Temps d'exécution de l'algorithme de numérotation

Tableau 4 – Observed gain with *b_comm++*

	amazon			dblp		
nb de cœurs	1 cœur	19 cœurs	32 cœurs	1 cœur	19 cœurs	32 cœurs
sans numérotation (ms)	6020869	384892	258333	3079606	190321	131220
avec numérotation (ms)	5631593	312224	204081	2922478	167365	112695
Gain (ms)	389276	72668	54252	157128	22956	18525
(n fois coût *NumBaCo*)	***232,4 fois***	***43,4 fois***	***32,4 fois***	***93,4 fois***	***13,6 fois***	***11 fois***

L'algorithme *NumBaCo* proposé ici peut aussi être utilisé pour le pré-processing dans des applications où le même graphe est utilisé pour plusieurs exécutions. C'est le cas par exemple des *benchmarks* pour lesquels plusieurs expérimentations sont effectuées avec le même graphe. Le graphe est alors traité et stocké uniquement à la première expérimentation. Dans ce cas, le coût de *NumBaCo*, même s'il est élévé par rapport au *benchmark*, est amortit par le nombre d'expérimentations.

Une autre utilisation de *NumBaCo* peut être perçue dans les plateformes de *streaming* comme Stinger [5]. Ici, on peut imaginer que le coût est élévé à l'initialisation du système et moins élévé durant le reste de la vie du système. Nous n'avons pas étudié ce cas dans cet article, mais au regard de la théorie developpée ici, le gain généré par *NumBaCo* permettrait d'augmenter les permeformances de ces plateformes.

5. Travaux connexes

Dans de récents travaux, Junya Arai et ses co-auteurs [1] se servent d'une modification de l'algorithme de Louvain pour proposer une numérotation basée aussi sur la structure en communautés des graphes. Toutefois, comparé à leurs travaux, dans notre article (qui étend [12]), nous restons les seuls :

– à avoir proposé un classement des communautés avant la numérotation, ce qui contribue à augmenter les performances ;

– à avoir proposé un stockage des voisins en suivant l'ordre de la structure imbriquée de Louvain (voir section 2.1.1), ce qui donne la priorité aux nœuds appartenant à la même communauté ou sous-communauté, augmentant ainsi les performances ;

– à avoir montré que l'algorithme proposé est une heuristique d'un problème plus général, le problème de numérotation des graphes sociaux. Nous avons aussi montré que ce problème est NP-complet.

D'autres auteurs se sont aussi servis de la structure en communautés des graphes pour une bonne organisation des données :

- Duong et co-auteurs [4] formalisent le problème de répartition des réseaux sociaux sur un système distribué de machines. Ils proposent ensuite un algorithme de répartition qui tire profit de la structure en communautés des réseaux sociaux. Leur objectif est de réduire le nombre de requêtes à la base de données. Dans notre cas, nous recherchons un algorithme de numérotation des nœuds dans une mémoire (partagée) et permettant de réduire les défauts de cache.

- Hoque et co-auteurs [9] ont conçu une technique d'organisation du disque dur en se basant sur le regroupement en communautés des données issues des graphes sociaux. Cette technique leur a permis de diminuer le nombre de déplacements de la tête de lecture et ainsi d'améliorer l'accès au disque (48% plus rapide). Nous agissons plutôt sur la mémoire vive (tandis qu'ils agissent sur le disque dur) ; et nous cherchons à réduire le nombre de défauts de cache.

Pour améliorer le *prefetching*, Li et co-auteurs [11] utilisent l'algorithme de recherche des itemsets (fréquents fermés) pour fabriquer leurs propres algorithmes (c-miner et c-miner*). Ces algorithmes sont ensuite utilisés pour trouver la corrélation entre les blocs d'une mémoire : les blocs sont perçus comme des items, les règles d'association issues de ces items permettent de faire du *prefetching*. Dans notre cas, nous contribions aussi à améliorer le *prefetching* mais en se servant de la détection des communautés.

Plusieurs travaux visent l'usage d'une meilleure représentation des matrices creuses pour accroitre les performances de certaines applications (dans la résolution des systèmes d'équations linéaires) :

- Lukas Polok et co-auteurs [16] proposent une structure de données basée sur la représentation en sous-blocs d'une matrice creuse. Cette représentation permet de réduire les défauts de cache lors des opérations arithmétiques effectuées sur la matrice pendant l'exécution.

- Rukhsana S. et Anila U. [18] proposent une représentation en sous-blocs (d'éléments tous non nuls) d'une matrice creuse. Les auteurs montrent que, non seulement cette représentation permet d'économiser plus d'espace, mais aussi permet d'obtenir une meilleur perfomance (lors de la multiplication matrice-vecteur).

Dans notre cas, nous recherchons la représentation mémoire des graphes sociaux la plus appropriée pour réduire les défauts de cache des programmes d'analyse des réseaux sociaux.

6. Conclusion

Dans cet article, il était question de voir comment exploiter la structure en communautés pour diminuer les défauts de cache et le temps d'exécution des programmes de fouille des réseaux sociaux. Nous avons proposé *NumBaCo*, une numérotation des nœuds permettant de tenir compte de la struture en communautés des graphes sociaux. Des expérimentations effectuées sur le score de katz et le Pagerank avec les jeux de données amazon, dblp et web-google ont montré que les performances sont améliorées lorsqu'on utilise cette numérotation.

En perspective, nous envisageons d'intégrer cet algorithme dans les langages dédiés et les plates-formes d'analyse de graphes afin d'améliorer leurs performances. Une autre perspective est de se servir de la structure en communautés des graphes sociaux lors de l'optimisation des boucles (pendant la phase de compilation) ; ceci pourrait contribuer à augmenter les performances.

7. Bibliographie

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.

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.

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.

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.

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

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

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

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.

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

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

Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS)*, 1(2) :213–245, 2005.

Thomas Messi-Nguélé, Maurice Tchuenté, and Jean-François Mehaut. Exploitation de la structure en communautés pour la réduction de défauts de cache dans la fouille des réseaux sociaux. In *Conférence de Recherche en Informatique (CRI)*, Yaoundé, Cameroon, December 2015.

Blaise Ngonmang, Maurice Tchuente, and Emmanuel Viennet. Local community identification in social networks. *Parallel Processing Letters*, 22(01) :1240004, 2012.

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.

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

Lukas Polok, Viorela Ila, and Pavel Smrz. Cache efficient implementation for block matrix operations. In *Proceedings of the High Performance Computing Symposium*, page 4. Society for

Computer Simulation International, 2013.

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. IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

Rukhsana Shahnaz and Anila Usman. Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers. *Int. Arab J. Inf. Technol.*, 8(2) :130–136, 2011.

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

F.3 Exploitation De La Structure En Communauté Pour la Réduction Des Défauts de Cache Dans La Fouille Des Réseaux Sociaux

EXPLOITATION DE LA STRUCTURE EN COMMUNAUTÉS POUR LA RÉDUCTION DES DÉFAUTS DE CACHE DANS LA FOUILLE DES RÉSEAUX SOCIAUX

Thomas Messi Nguélé[1,2,3] *Maurice Tchuente*[1,2] *Jean-Francois Méhaut*[2,3]

[1] *IRD, UMI 209, UMMISCO, Université de Yaoundé I BP 337 Yaoundé Cameroun*

[2] *LIRIMA, Laboratoire d'Informatique et Applications, Fac. des Sciences,Dépt. d'informatique, BP: 812 Yaoundé, Cameroun*

[3] *Université de Grenoble, INRIA-LIG, Corse BP: 38400 Grenoble, France*

ABSTRACT

One of social graphs properties is the community structure, that is subsets where nodes beloging to the same subset have a higher link density and a low link density with nodes beloging to external subset. Otherwise, most social network mining algorithms have a local exploration of the underlying gaph, which leads to start with a node and make reference to the neighborhood of this node. The idea of this paper is to exploit the community structure for optimise the storage of large graphs that arise in social network mining. The goal of such a data structure is to reduce cache misses with consequent reduction of execution time. In this paper, we present for Katz score, simulation exploiting community structure product by Louvain algorithm. Results on a NUMA32 machine show that, comparing with others representations including Yale, adjacency list or bloc (representations used in graph analysis platforms like Green-Marl, Galois or Stinger), we obtain gains up to 20% for cache misses and 14% for execution time.

Keywords : *Social network mining, Community, Cache miss.*

RESUME

L'une des propriétés des graphes sociaux est leur structure en communautés, c'est à dire en sous-ensembles où les nœuds ont une forte densité de liens entre eux et une faible densité de liens avec l'extérieur. Par ailleurs, la plupart des algorithmes de fouille des réseaux sociaux comportent une exploration locale du graphe sous-jacent ce qui amène à partir d'un nœud à faire référence aux nœuds situés dans son voisinage. L'idée de cet article est d'exploiter la structure en communautés pour optimiser le stockage des grands graphes qui surviennent dans la fouille des réseaux sociaux. L'objectif d'une telle structure est de réduire le nombre de défauts de cache avec pour conséquence la réduction du temps d'exécution. Dans cet article, nous présentons pour le score de Katz, des simulations avec un stockage exploitant la structure en communautés produite par l'algorithme de Louvain. Les résultats sur une machine NUMA32 montrent que, par rapport aux représentations de Yale, par liste ou par blocs d'adjacence (représentations utilisées dans des plates-formes d'analyse de graphes comme Green-Marl, Galois ou Stinger), on obtient des gains pouvant aller jusqu'à 20% pour les défauts de cache et 14% pour le temps d'exécution.

Mots clés : *Fouille de réseaux sociaux, Communauté, Défaut de cache.*

CRI-2015, December. 14 - 15, 2015, Yaounde, Cameroon.

1. INTRODUCTION

Lorsqu'un algoritme de fouille des réseaux sociaux (comme par exemple le calcul du score de Katz [6]) s'execute, il opère sur chaque nœud x du graphe en faisant le plus souvent référence aux nœuds situés dans le voisinage de x. Les structures de données utilisées dans les DSLs de graphes ou les plates-formes d'analyse des graphes rencontrés dans l'état de l'art (Galois [8], Green-Marl [4], Stinger [1, 2] ...) ne permettent pas d'exploiter cette organisation des données des graphes sociaux pendant l'exécution.

Dans ce rapport, nous nous intéressons à l'impact de l'organisation en mémoire des données de graphes sociaux sur la réduction du temps d'exécution des programmes. En d'autres termes, peut-on réduire les défauts de cache et donc le temps d'exécution des algorithmes des graphes sociaux si l'on tient compte, dans la structure de données utilisée dans le runtime, de l'organisation en communauté des nœuds du graphe?

1.1. Exemple introductif

Considérons le graphe (G) non orienté de la figure 1 ayant 16 nœuds répartis en quatre communautés.

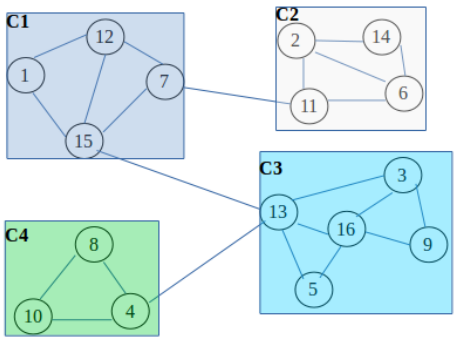


Figure 1: Graphe (G). Les Ci représentent les communautés

Une façon simple de représenter ce graphe est d'utiliser la représentation matricielle (voir Table 1). Mais celle-ci n'est pas très appropriée pour les graphes sociaux car les matrices resultantes sont souvent très creuses:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1												1			1	
2						1					1			1		
3									1				1			1
4								1		1			1			
5													1			1
6		1									1			1		
7											1	1			1	
8				1						1						
9			1													1
10				1				1								
11		1				1	1									
12	1						1								1	
13			1	1	1										1	1
14		1				1										
15	1						1					1	1			
16			1		1				1				1			

Table 1: Matrice représentant le graphe (G)

La représentation souvent adoptée par certains DSLs de graphe (à l'instard de Galois [8] et Green-Marl [4]) est celle de Yale [3]. Dans cette représentation, on simule la matrice représentant le graphe avec trois vecteurs :

- un vecteur représentant les arêtes (vecteur A), chaque arête étant représentée par une de ses extrémités,
- un autre vecteur donnant l'autre extrémité de chacune des arêtes de A (vecteur JA),
- et un dernier vecteur qui donne l'indice dans le vecteur initial du premier élément non nul de chaque ligne de la matrice simulée (vecteur IA).

D'autres représentations peuvent être utilisées: le graphe est alors représenté par un tableau de nœuds, chaque nœud pouvant être relié

1. à une liste chainée de ses voisins
2. à un bloc de ses voisins, la taille du bloc étant variable (utilisé dans [10])
3. à une liste chainée de blocs (de taille fixe) de ses voisins (adapté aux gaphes dynamiques, utilisé par la plateforme Stinger [1, 2]).

La représentation de Yale de l'exemple précédent est la suivante:

A	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
–	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

JA	12	15	6	11	14	9	13	16	8	10	13	13	16	2	11	14	11	12	15	4	10	3	16
–	4	8	2	6	7	1	7	15	3	4	5	15	16	2	6	1	7	12	13	3	5	9	13

IA	1	3	6	9	12	14	17	20	22	24	26	29	32	37	39	43	47

Aucune de ces représentations ne tire profit du "regroupement en communauté" des nœuds du graphe pour réduire le temps d'exécution des algorithmes des réseaux sociaux, car ce n'était pas leur but.

2. COMMENT EXPLOITER LA STRUCTURE EN COMMUNAUTÉS DES GRAPHES SOCIAUX POUR DIMINUER LES DÉFAUTS DE CACHE?

2.1. Idée

L'idée est de faire en sorte que, chaque fois qu'un nœud est un cours de traitement, les autres nœuds membres de sa communauté se retrouvent dans le même cache mémoire que ce nœud. Ainsi, les données correspondant à une communauté doivent être consécutives en mémoire.

Pour atteindre cet objectif, nous représentons une communauté par un tableau de ses nœuds membres. Chaque nœud est relié à un tableau de ses voisins (les voisins qui ne sont pas membres de la communauté sont classés au fond du tableau, ceci aura pour effet de donner la priorité aux membres de la communauté lors d'un traitement). Par exemple, à la figure 2, la communauté C1 est constituée d'un tabeau de 4 éléments (1,7,12,15); les voisins 11 du nœud 7 et 13 du nœud 15 sont classés au fond des tableaux des voisins.

2.2. Gestion du cache

Deux cas de figure sont envisagés.

2.2.1. Cas des processeurs généralistes

Les processeurs généralistes tels que Intel, AMD, ARM implémentent dans le matériel leur propre algorithme de gestion de cache. Ainsi, pour bénéficier du regroupement en communauté des nœuds, le calcul des communautés est effectué au moment du chargement du graphe en mémoire. Le graphe est alors représenté par un tableau de communautés. Les communautés sont classées par dégré d'affinité (plus deux communautés partagent un grand nombre de liens, plus il y a de chance que ces communautés soient consécutives dans le tableau). La figure 2 montre la représentation mémoire du graphe (G).

Lorsque survient un défaut de cache, les algorithmes de remplacement de ligne de cache s'exécutent alors sans changer l'organisation en communauté dans la structure de données.

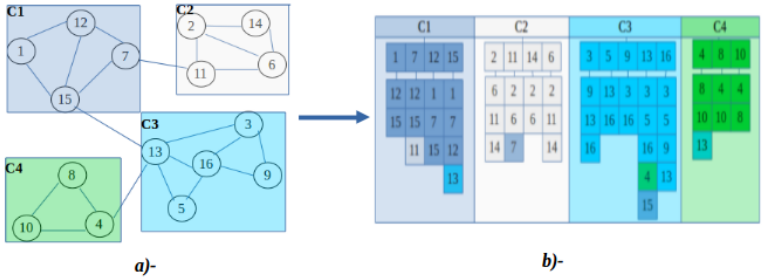


Figure 2: Représentation de (G) tenant compte des communautés

2.2.2. Cas des processeurs laissant la gestion du cache au programmeur

C'est le cas par exemple du processeur MPPA de Kalray où la mémoire locale d'un cluster peut-être perçue comme un cache. Ici, lorsque survient un défaut de cache, il y a un calcul de la communauté (de la donnée correspondant au nœud ayant créé le défaut de cache). Cette communauté est alors chargée dans le cache en suivant l'un des algorithmes classiques:

- algorithme optimal (la ligne de cache qui ne sera pas utilisée pour la plus grande période de temps est remplacée),
- algorithme aléatoire,
- LRU *Least Recently Used*, - FIFO *First In First Out*,
- LFU *Least Frequently Used*, ...

Ici, il peut arriver que la taille de la communauté soit très grande ou très petite. Dans le cas où elle est grande, on pourra se contenter de la communauté locale (ou d'un autre moyen permettant de réduire cette taille). Dans le cas où elle très petite, on chargera aussi la communauté ayant le plus de liens avec la communauté initiale. (On se retrouve ici avec un problème du sac à dos particulier: les éléments ne sont pas indépendants).

2.3. Modèle mathématique

3. TRAVAUX CONNEXES

Hoque et co-auteurs [5] ont conçu une technique d'organisation du disque dur en se basant sur le regroupement en communautés des données issues des graphes sociaux. Cette technique leur a permis de diminuer le nombre de déplacements de la tête de lecture et ainsi d'améliorer l'accès au disque (48% plus rapide). Nous agissons plutôt sur la mémoire vive (tandis qu'ils agissent sur le disque dur).

Pour améliorer le *prefetching*, Li et co-auteurs [7] utilisent l'algorithme de recherche des itemsets (fréquents fermés) pour fabriquer leurs propres algorithmes (c-miner et c-miner*). Ces algorithmes sont ensuite utilisés pour trouver la corrélation entre les blocs d'une mémoire: les blocs sont perçus comme des items, les règles d'association issues de ces items permettent de faire du *prefetching*. Dans notre cas, nous nous servons de la détection des communautés.

Plusieurs travaux visent l'usage d'une meilleur représentation des matrices creuses pour accroitre les performances de certaines applications (dans la résolution des systèmes d'équations linéaires):

- Lukas Polok et co-auteurs [9] proposent une structure de données basée sur la représentation en sous-blocs d'une matrice creuse. Cette représentation permet de réduire les défauts de cache lors des opérations arithmétiques effectuées sur la matrice pendant l'exécution.

- Rukhsana S. et Anila U. [11] proposent une représentation en sous-blocs (d'éléments tous non nuls) d'une matrice creuse. Les auteurs montrent que, non seulement cette représentation permet d'économiser plus d'espace, mais aussi permet d'obtenir une meilleur perfomance (lors de la multiplication matrice-vecteur).

Dans notre cas, nous recherchons la structure de données la plus appropriée pour les programmes de la fouille des réseaux sociaux.

4. ÉVALUATION

Les expérimentations ont été menées sur la machine NUMA32, 4 nœuds de 8 cœurs chacun, soit au total 32 cœurs pour 64 Go. Chaque noeud est de type Intel Xeon avec les caractéristiques 2.27GHz, L1 32KB, L2 256KB, L3 24MB, pas d'*Hyper-Threading*.

Pour cette évaluation, nous avons implémenté l'algorithme du score de katz [6] avec quatre structures de données pour la repésentation des graphes:

- représentation du graphe en exploitant la structure en communauté des nœuds *(comm_ds)*
- représentation du graphe avec un tableau de nœuds, chacun étant relié à une liste chainée de ses voisins *(list_ds)*
- représentation du graphe avec un tableau de nœuds, chacun étant relié à un bloc de ses voisins, la taille du bloc étant variable *(bloc_ds)*
- représentation du graphe sous forme de Yale *(yale_ds)*

4.1. Algorithme du score de katz

Le score de Katz [6] est utilisé comme une mesure de similarité basée sur les distances entre les nœuds. Le score de Katz entre deux nœuds x et y est donné par la formule:

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

Où:

- L représente la taille maximale d'un chemin.
- $paths_{x,y}^{<l>}$ est l'ensemble des chemins de longueur l entre x et y, et $|paths_{x,y}^{<l>}|$ représente leur nombre.
- $0 < \beta < 1$. β est choisi tel que les chemins avec un l grand contribuent moins à la somme que les chemins avec un l petit.
- $A^l(x,y) = paths_{x,y}^{<l>}$, A étant la matrice d'adjacence.

Nous avons réalisé une implémentation multithreadée de l'algorihme de katz. Les nœuds sont rangés dans une liste chainée qui est ensuite parcourue en parallèle. Pour chaque nœud, la fonction **computeKatzNode** est invoquée (voir algorithme 1).

Algorithm 1 Katz multi-threadé

```
1:  Global nodeList, G, β, L, K_sc
2:  do_work()
3:  while nodeList ≠ ∅ do
4:      x ← atomic_dequeue(nodeList)
5:      K_sc[x] ← computeKatzNode(x, G, β, L)
6:  end while
7:
8:  main()
9:  nodeList ← generate_nodeList(G)
10: for i = 1 to n_threads do
11:     spawn_thread(do_work())
12: end for
13: wait_every_child_thread()
14: output(K_sc)
```

Pour tout nœud x de G, on peut démontrer que les nombres de chemins à partir de ce nœud vers les autres nœuds se calculent ainsi qu'il suit (N_i, et L_i représentent respectivement les voisins et les nombres de chemins d'ordre i):

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

Ceci nous permet d'établir l'algorithme 2. La clé réside dans le calcul du vecteur de nombre de chemins $cLenPath$ à la ligne 6 avec la fonction **updateLenPath**() développée entre les lignes 16 et 27).

Algorithm 2 Score de Katz entre x et tout nœud atteignable avec L

```
1:  computeKatzNode(x, G, β, L)
2:  dNeig ← G.neighbors(x)
3:  pNeig ← dNeig
4:  pLenPath[{dNeig}] ← 1
5:  for l = 2 → L do
6:     [cNeig, cLenPath] ← updateLenPath(pNeig[ ], pLenPath[ ])
7:     for all (t ∈ cNeig) and (t ∉ dNeig) do
8:        katz[t] ← katz[t] + β^l cLenPath[t]
9:        accessibleNeig.add(t)
10:    end for
11:    [pNeig, cNeigh] ← [cNeig, empty()]
12:    [pLenPath, cLenPath] ← [cLenPath, empty()]
13: end for
14: return buildLign(x, katz[ ], accessibleNeig[ ])
15:
16: updateLenPath(pNeig[ ], pLenPath[ ])
17: for all y ∈ pNeig do
18:    for all z ∈ G.neighbors(y) do
19:       if z ∈ cNeig then
20:          cLenPath[z] ← cLenPath[z] + pLenPath[z]
21:       else
22:          cLenPath[z] ← pLenPath[z]
23:          cNeig.add(z)
24:       end if
25:    end for
26: end for
27: return [cNeig, cLenPath]
```

À chaque valeur de l, avant de passer à la valeur suivante, le score de katz est mis à jour (lignes 8 à 11); l'ensemble des nœuds courants et le tableau des nombres de chemins sont également mis à jour (lignes 12 et 13).

4.2. Résultats

L'impact du regroupement des nœuds dans chacune des structures de données est visible. En effet, à la figure 3 jusqu'à 20 cœurs, le temps obtenu avec la structure *comm_ds* reste inférieur à celui obtenu avec les autres structures (avec 1 cœur par exemple, on a 245 s de différence soit environ 14%). Ceci montre bien que le fait que les nœuds soient regroupés en communautés contribue à réduire le temps d'exécution.

Entre 20 cœurs et 32 cœurs, les temps d'exécutions deviennent pratiquement identiques pour toutes les structures de données. Le gain en parallélisme n'est plus très significatif. Ceci peut s'expliquer par le fait qu'à ce niveau, la plus grande partie du temps d'exécution provient des synchronisations entre threads: les threads passent plus de temps dans la synchronisation. Ainsi l'influence de la structure de données comm_ds n'est plus perceptible.

En augmentant la longueur maximale d'un chemin, (le paramètre *L* passe de 2 à 3), la granularité de la tâche affectée à chaque thread est augmentée. Ceci a pour effet de minimiser le temps de synchronisation. On peut ainsi observer, à la figure 4, que le temps obtenu avec la structure *comm_ds* reste plus petit que les temps d'exécution obtenus avec les autres. Ceci confirme bien que le regroupement en communautés des nœuds contribue à diminuer le temps d'exécution.

Pour voir si les temps d'exécution observés étaient liés au nombre de défauts de caches causés par le programme, nous avons lancé le programme avec l'outil *perf* [1]. Le tableau 2 nous présente par exemple les résutats obtenus (avec 1 cœur) pour les évenements "cache-misses" et "L1-dcache-load-misses". Nous pouvons ainsi observer que la struture en commuauté nous permet de diminuer environ 20% de défauts de cache.

[1] `https://perf.wiki.kernel.org/index.php/Tutorial`

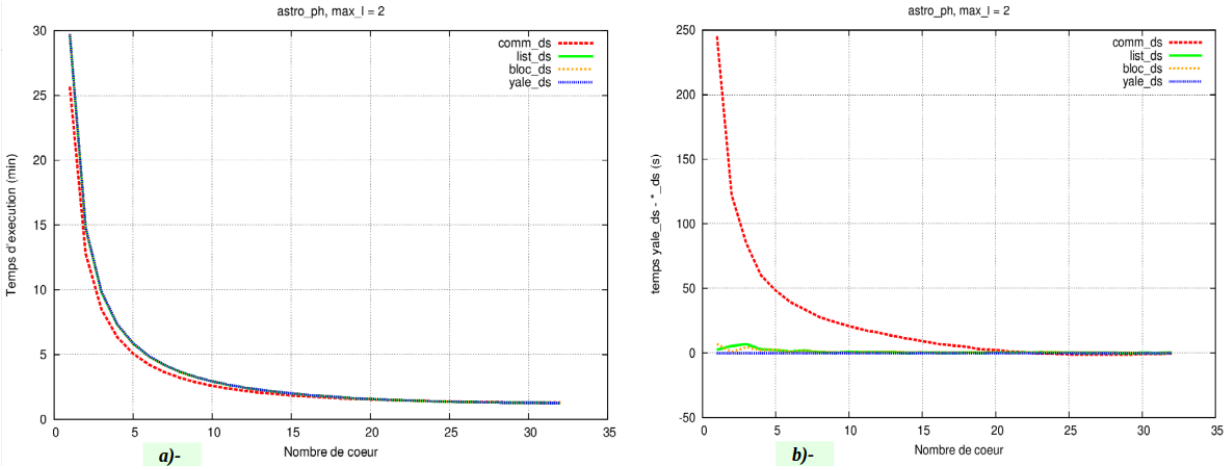


Figure 3: Temps d'exécution en fonction du nombre de cœurs. À droite, on a la différence de temps avec yale_ds

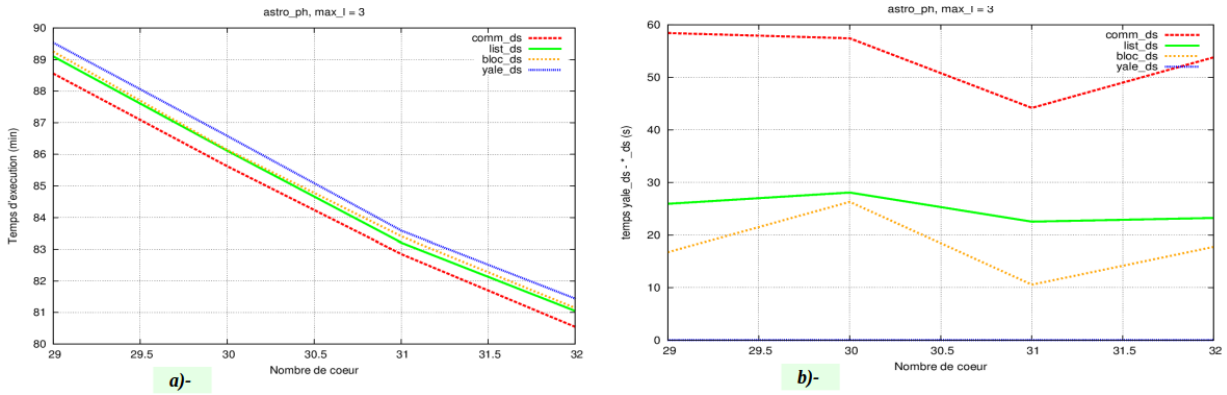


Figure 4: À gauche, on a la différence de temps avec yale_ds (à partir de 29 cœurs)

5. CONCLUSION

Dans cet article, il était question de voir comment exploiter la structure en communautés pour diminuer les défauts de cache et le temps d'exécution des programmes de fouille des réseaux sociaux. Nous avons proposé une structure de données permettant de tenir compte de cette struture en communautés des graphes sociaux. Des expérimentations effectuées sur le score de katz ont montré que les défauts de caches et le temps d'exécution sont réduits lorsqu'on utilise cette structure de données.

En perspective, il sera intéressant de faire des évaluations avec d'autres applications des graphes tels que: *betweenness centrality, pagerank, calcul des plus courts chemins*. Il faudra ensuite de démontrer à l'aide d'un modèle mathématique les resultats observés. Il sera enfin intéressant d'implémenter cette structure de données dans les DSLs et les plates-formes d'analyse de graphes afin d'améliorer leurs performances.

6. REFERENCES

[1] 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.

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

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

évenement	comm_ds	list_ds	bloc_ds	yale_ds
cache-misses(millions)	8,243	10,469	10,128	10,458
L1-dcache-load-misses(millions)	63,802	72,608	72,573	72,528

Table 2: Nombre de défauts de cache

[4] 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.

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

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

[7] Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS)*, 1(2):213–245, 2005.

[8] 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.

[9] Lukas Polok, Viorela Ila, and Pavel Smrz. Cache efficient implementation for block matrix operations. In *Proceedings of the High Performance Computing Symposium*, page 4. Society for Computer Simulation International, 2013.

[10] 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. IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

[11] Rukhsana Shahnaz and Anila Usman. Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers. *Int. Arab J. Inf. Technol.*, 8(2):130–136, 2011.

Appendix G

Liste Protocolaire

UNIVERSITÉ DE YAOUNDÉ I **Faculté des Sciences** Division de la Programmation et du Suivi des Activités Académiques		THE UNIVERSITY OF YAOUNDE I **Faculty of Science** Division of Programming and Follow-up of Academic Affaires
LISTE DES ENSEIGNANTS PERMANENTS		**LIST OF PERMANENT TEACHING STAFF**

ANNÉE ACADEMIQUE 2017/2018

(Par Département et par Grade)

DATE D'ACTUALISATION : 10 Mars 2018

ADMINISTRATION

DOYEN :AWONO ONANA Charles, *Professeur*
VICE-DOYEN / DPSAA :DONGO Etienne, *Professeur*
VICE-DOYEN / DSSE :OBEN Julius ENYONG,*Professeur*
VICE-DOYEN / DRC :MBAZE MEVA'A Luc Léonard, Maitre de Conférences
Chef Division Administrative et Financière : NDOYE FOE Marie C. F., *Maitre de Conférences*
Chef Division des Affaires Académiques, de la Scolarité et de la Recherche : ABOSSOLO Monique, *Maitre de Conférences*

1- DÉPARTEMENT DE BIOCHIMIE (BC) (41)			
N°	NOMS ET PRÉNOMS	GRADE	OBSERVATIONS
1	FEKAM BOYOM Fabrice	Professeur	En poste
2	MBACHAM FON Wilfried	Professeur	En poste
3	MOUNDIPA FEWOU Paul	Professeur	Chef de Département
4	NINTCHOM PENLAP V. épse BENG	Professeur	En poste
5	OBEN Julius ENYONG	Professeur	En poste
6	ATOGHO Barbara Mma	Maître de Conférences	En poste
7	BELINGA née NDOYE FOE M. C. F.	Maître de Conférences	Chef DAF / FS
8	BIGOGA DIAGA Jude	Maître de Conférences	En poste
9	BOUDJEKO Thaddée	Maître de Conférences	En poste
10	EFFA NNOMO Pierre	Maître de Conférences	En poste
11	FOKOU Elie	Maître de Conférences	En poste
12	KANSCI Germain	Maître de Conférences	En poste
13	NANA Louise épouse WAKAM	Maître de Conférences	En poste
14	NGONDI Judith Laure	Maître de Conférences	En poste
15	NGUEFACK Julienne	Maître de Conférences	En poste
16	NJAYOU Frédéric Nico	Maître de Conférences	En poste
17	ACHU Merci BIH	Chargée de Cours	En poste
18	BIYITI BI ESSAM née AKAM ADA L.	Chargée de Cours	CT MINRESI
19	DEMMANO Gustave	Chargé de Cours	En poste
20	DJOKAM TAMO Rosine	Chargée de Cours	En poste
21	DJUIDJE NGOUNOUE Marcelline	Chargée de Cours	En poste
22	DJUIKWO NKONGA Ruth Viviane	Chargée de Cours	En poste
23	EVEHE BEBANDOUE Marie-Solange	Chargée de Cours	En poste
24	EWANE Cécile Anne	Chargée de Cours	En poste
25	KOTUE KAPTUE Charles	Chargé de Cours	En poste
26	LUNGA Paul KEILAH	Chargé de Cours	En poste

27	MBONG ANGIE M. Mary Anne	Chargée de Cours	En poste
28	MOFOR née TEUGWA Clotilde	Chargée de Cours	CE SEP MINESUP
29	NJAYOU Frédéric Nico	Chargé de Cours	En poste
30	Palmer MASUMBE NETONGO	Chargé de Cours	En poste
31	TCHANA KOUATCHOUA Angèle	Chargée de Cours	En poste
32	PACHANGOU NSANGOU Sylvain	Chargé de Cours	En poste
33	DONGMO LEKAGNE Joseph Blaise	Chargé de Cours	En poste
34	FONKOUA Martin	Chargé de Cours	En poste
35	BEBOY EDZENGUELE Sara Nathalie	Chargée de Cours	En poste
36	DAKOLE DABOY Charles	Chargée de Cours	En poste
37	MANANGA Marlyse Joséphine	Chargée de Cours	En poste
38	MBOUCHE FANMOE Marceline Joëlle	Assistante	En poste
39	BEBEE Fadimatou	Assistante	En poste
40	TIENTCHEU DJOKAM Leopold	Assistant	En poste

2- DÉPARTEMENT DE BIOLOGIE ET PHYSIOLOGIE ANIMALES (BPA) (44)			
1	BILONG BILONG Charles-Félix	Professeur	**Chef de Département**
2	DIMO Théophile	Professeur	En Poste
3	DJIETO LORDON Champlain	Professeur	En poste
4	ESSOMBA née NTSAMA MBALA	Professeur	*VDoyen/FMSB/UYI*
5	FOMENA Abraham	Professeur	En Poste
6	KAMTCHOUING Pierre	Professeur	EN POSTE
7	NJAMEN Dieudonné	Professeur	En poste
8	NJIOKOU Flobert	Professeur	En Poste
9	NOLA Moïse	Professeur	En poste
10	TAN Paul VERNYUY	Professeur	En poste
11	TCHUEM TCHUENTE Louis Albert	Professeur	*Coord. Progr. MINSANTE*
12	AJEAGAH Gidéon AGHAINDUM	Maître de Conférences	**Chef Service DPER**
13	DZEUFIET DJOMENI Paul Désiré	Maître de Conférences	En poste
14	FOTO MENBOHAN Samuel	Maître de Conférences	En poste
15	KAMGANG René	Maître de Conférences	*C.S. MINRESI*
16	KEKEUNOU Sévilor	Maître de Conférences	En poste
17	MEGNEKOU Rosette	Maître de Conférences	En poste
18	MONY Ruth épse NTONE	Maître de Conférences	En Poste
19	TOMBI Jeannette	Maître de Conférences	En poste
20	ZEBAZE TOGOUET Serge Hubert	Maître de Conférences	En poste
21	ALENE Désirée Chantal	Chargée de Cours	En poste
22	ATSAMO Albert Donatien	Chargée de Cours	En poste
23	BELLET EDIMO Oscar Roger	Chargé de Cours	En poste
24	BILANDA Danielle Claude	Chargée de Cours	En poste
25	DJIOGUE Séfirin	Chargée de Cours	En poste
26	DONFACK Mireille	Chargée de Cours	En poste
27	GOUNOUE KAMKUMO Raceline	Chargée de Cours	En poste
28	LEKEUFACK FOLEFACK Guy B.	Chargé de Cours	En poste
29	MAHOB Raymond Joseph	Chargé de Cours	En poste
30	MBENOUN MASSE Paul Serge	Chargé de Cours	En poste
31	MOUNGANG LucianeMarlyse	Chargée de Cours	En poste
32	MVEYO NDANKEU Yves Patrick	Chargée de Cours	En poste

33	NGOUATEU KENFACK Omer Bébé	Chargé de Cours	En poste
34	NGUEGUIM TSOFACK Florence	Chargée de Cours	En poste
35	NGUEMBOK	Chargé de Cours	En poste
36	NJATSA Hermine épse MEGAPTCHE	Chargée de Cours	En Poste
37	NJUA Clarisse Yafi	Chargée de Cours	**CD/UBa**
38	NOAH EWOTI Olive Vivien	Chargée de Cours	En poste
39	TADU Zephyrin	Chargée de Cours	En poste
40	YEDE	Chargée de Cours	En poste
41	ETEME ENAMA Serge	Assistant	En poste
42	KANDEDA KAVAYE Antoine	Assistant	En poste
43	KOGA MANG DOBARA	Assistant	En poste

3- DÉPARTEMENT DE BIOLOGIE ET PHYSIOLOGIE VÉGÉTALES (BPV) (26)

1	AMBANG Zachée	Professeur	Chef Division/UYII
2	BELL Joseph Martin	Professeur	En poste
3	YOUMBI Emmanuel	Professeur	**Chef de Département**
4	MOSSEBO Dominique Claude	Professeur	En poste
5	BIYE Elvire Hortense	Maître de Conférences	En poste
6	DJOCGOUE Pierre François	Maître de Conférences	En poste
7	KENGNE NOUMSI Ives Magloire	Maître de Conférences	En poste
8	MALA Armand William	Maître de Conférences	En poste
9	NDONGO BEKOLO	Maître de Conférences	*CE / MINRESI*
10	NGONKEU MAGAPTCHE Eddy L.	Maître de Conférences	En poste
11	ZAPFACK Louis	Maître de Conférences	En poste
12	MBARGA BINDZI Marie Alain	Maître de Conférences	CT/Univ Dschang
13	MBOLO Marie	Maître de Conférences	En poste
14	ANGONI Hyacinthe	Chargée de Cours	En poste
15	MAHBOU SOMO TOUKAM. Gabriel	Chargé de Cours	En poste
16	ONANA JEAN MICHEL	Chargé de Cours	En poste
17	GOMANDJE Christelle	Chargée de Cours	En poste
18	NGODO MELINGUI Jean Baptiste	Chargé de Cours	En poste
19	NGALLE Hermine BILLE	Chargée de Cours	En poste
20	NGOUO Lucas Vincent	Chargé de Cours	En poste
21	NSOM ZAMO Annie Claude épse PIAL	Chargée de Cours	*Expert national /UNESCO*
22	TONFACK Libert Brice	Chargé de Cours	En poste
23	TSOATA Esaïe	Chargé de Cours	En poste
24	DJEUANI Astride Carole	Assistante	En poste
25	MAFFO MAFFO Nicole Liliane	Assistante	En poste
26	NNANGA MEBENGA Ruth Laure	Assistante	En poste
27	NOUKEU KOUAKAM Armelle	Assistante	En poste

4- DÉPARTEMENT DE CHIMIE INORGANIQUE (CI) (33)

1	AGWARA ONDOH Moïse	Professeur	*Vice RecteurUniv ,Bamenda*
2	ELIMBI Antoine	Professeur	En poste
3	Florence UFI CHINJE épouse MELO	Professeur	*RECTEUR Univ.Ngaoundere*
4	GHOGOMU Paul MINGO	Professeur	*Directeur Cabinet PM*
5	LAMINSI Samuel	Professeur	En poste
6	NANSEU Charles Péguy	Professeur	En poste
7	NDIFON Peter TEKE	Professeur	*IS1 MINRESI/**Chef de Departement***
8	NENWA Justin	Professeur	En poste

9	NGAMENI Emmanuel	Professeur	*DOYEN FS Univ. Dschang*
10	BABALE née DJAM DOUDOU	Maître de Conférences	*Chargée Mission P.R.*
11	DJOUFAC WOUMFO Emmanuel	Maître de Conférences	En poste
12	KEMEGNE MBOUGUEM Jean C.	Maître de Conférences	En poste
13	KONG SAKEO	Maître de Conférences	*Chargé de Mission au P. M.*
14	NDIKONTAR Maurice KOR	Maître de Conférences	*Vice-Doyen Univ. Bamenda*
15	NGOMO Horace MANGA	Maître de Conférences	*VC/UB*
16	NJIOMOU C. épse DJANGANG	Maître de Conférences	En poste
17	YOUNANG Elie	Maître de Conférences	En poste
18	ACAYANKA Elie	Chargé de Cours	En poste
19	EMADACK Alphonse	Chargé de Cours	En poste
20	KAMGANG YOUBI Georges	Chargé de Cours	En poste
21	NDI NSAMI Julius	Chargée de Cours	En poste
22	NJOYA Dayirou	Chargé de Cours	En poste
23	PABOUDAM GBAMBIE A.	Chargée de Cours	En poste
24	TCHAKOUTE KOUAMO Hervé	Chargé de Cours	En poste
25	BELIBI BELIBI Placide Désiré	Chargé de Cours	En poste
26	CHEUMANI YONA Arnaud M.	Chargé de Cours	En poste
27	NYAMEN Linda Dyorisse	Chargée de Cours	En poste
28	KENNE DEDZO GUSTAVE	Chargé de Cours	En poste
29	KOUOTOU DAOUDA	Chargé de Cours	En poste
30	MAKON Thomas Beauregard	Chargé de Cours	En poste
31	MBEY Jean Aime	Chargé de Cours	En poste
32	NCHIMI NONO KATIA	Chargé de Cours	En poste
33	NEBA nee NDOSIRI Bridget NDOYE	Chargé de Cours	En poste
5- DÉPARTEMENT DE CHIMIE ORGANIQUE (CO) (34)			
1	DONGO Etienne	Professeur	**Vice-Doyen / DSSE**
2	GHOGOMU TIH Robert Ralph	Professeur	En poste
3	MBAFOR Joseph	Professeur	En poste
4	NGADJUI TCHALEU B.	Professeur	*Chef de Dépt FMBS*
5	NGOUELA Silvère Augustin	Professeur	En poste
6	NKENGFACK Augustin Ephraïm	Professeur	**Chef de Département**
7	NYASSE Barthélemy	Professeur	*Directeur/UN*
8	PEGNYEMB Dieudonné Emmanuel	Professeur	*Directeur/ MINESUP*
9	WANDJI Jean	Professeur	En poste
10	Alex de Théodore ATCHADE	Maître de Conférences	*CD Rectorat/UYI*
11	FOLEFOC Gabriel NGOSONG	Maître de Conférences	*Vice-Doyen Univ. Buea*
12	KEUMEDJIO Félix	Maître de Conférences	En poste
13	KOUAM Jacques	Maître de Conférences	En poste
14	MBAZOA née DJAMA Céline	Maître de Conférences	En poste
15	NOUNGOUE TCHAMO Diderot	Maître de Conférences	En poste
16	TCHOUANKEU Jean-Claude	Maître de Conférences	*VR/ UYII*
17	YANKEP Emmanuel	Maître de Conférences	En poste
18	TIH née NGO BILONG E. Anastasie	Maître de Conférences	En poste
19	MKOUNGA Pierre	Maître de Conférences	En poste
20	NGO MBING Joséphine	Maître de Conférences	En poste
21	TABOPDA KUATE Turibio	Maître de Conférences	En poste
22	KEUMOGNE Marguerite	Maître de Conférences	En poste
23	AMBASSA Pantaléon	Chargé de Cours	En poste
24	EYONG Kenneth OBEN	Chargé de Cours	En poste
25	FOTSO WABO Ghislain	Chargé de Cours	En poste

26	KAMTO Eutrophe Le Doux	Chargé de Cours	En poste
27	NGONO BIKOBO Dominique Serge	Chargé de Cours	En poste
28	NOTE LOUGBOT Olivier Placide	Chargé de Cours	En poste
29	OUAHOUO WACHE Blandine M.	Chargée de Cours	En poste
30	TAGATSING FOTSING Maurice	Chargé de Cours	En poste
31	ZONDENDEGOUMBA Ernestine	Chargée de Cours	En poste
32	NGOMO Orléans	Chargée de Cours	En poste
33	NGNINTEDO Dominique	Assistant	En poste
6- DÉPARTEMENT D'INFORMATIQUE (IN) (25)			
1	ATSA ETOUNDI Roger	Professeur	**Chef de Département**
2	FOUDA NDJODO Marcel Laurent	Professeur	*Chef Dpt ENS/Chef DivSys.MINESUP*
3	TCHUENTE Maurice	Professeur	*PCA UB*
4	NDOUNDAM Réné	Maître de Conférences	En poste
5	KOUOKAM KOUOKAM E. A.	Chargé de Cours	En poste
6	CHEDOM FOTSO Donatien	Chargé de Cours	En poste
7	MELATAGIA YONTA Paulin	Chargé de Cours	En poste
8	MOTO MPONG Serge Alain	Chargé de Cours	En poste
9	TINDO Gilbert	Chargé de Cours	En poste
10	TSOPZE Norbert	Chargé de Cours	En poste
11	WAKU KOUAMOU Jules	Chargé de Cours	En poste
12	TAPAMO Hyppolite	Chargé de Cours	En poste
13	ABESSOLO ALO'O Gislain	Assistant	En poste
14	BAYEM Jacques Narcisse	Assistant	En poste
15	DJOUWE MEFFEJA Merline Flore	Assistante	En poste
16	DOMGA KOMGUEM Rodrigue	Assistant	En poste
17	EBELE Serge	Assistant	En poste
18	HAMZA Adamou	Assistant	En poste
19	KAMDEM KENGNE Christiane	Assistante	En poste
20	KAMGUEU Patrick Olivier	Assistant	En poste
21	KENFACK DONGMO Clauvice V.	Assistant	En poste
22	MEYEMDOU Nadège Sylvianne	Assistante	En poste
23	MONTHE DJIADEU Valery M.	Assistant	En poste
24	JIOMEKONG AZANZI Fidel	Assistant	En poste
7- DÉPARTEMENT DE MATHÉMATIQUES (MA) (35)			
1	BEKOLLE David	Professeur	*Vice-Recteur UN*
2	BITJONG NDOMBOL	Professeur	*En poste*
3	DOSSA COSSY Marcel	Professeur	En poste
4	AYISSI Raoult Domingo	Maître de Conférences	**Chef de Département**
5	EMVUDU WONO Yves S.	Maître de Conférences	*CD/ MINESUP*
6	NKUIMI JUGNIA Célestin	Maître de Conférences	En poste
7	NOUNDJEU Pierre	Maître de Conférences	En poste
8	TCHAPNDA NJABO Sophonie B.	Maître de Conférences	Directeur/AIMS Rwanda
9	AGHOUKENG JIOFACK Jean Gérard	Chargé de Cours	Chef Service MINPLAMAT
10	CHENDJOU Gilbert	Chargé de Cours	En poste
11	FOMEKONG Christophe	Chargé de Cours	En poste
12	KIANPI Maurice	Chargé de Cours	En poste
13	KIKI Maxime Armand	Chargé de Cours	En poste
14	MBAKOP Guy Merlin	Chargé de Cours	En poste
15	MBANG Joseph	Chargé de Cours	En poste
16	MBEHOU Mohamed	Chargé de Cours	En poste

17	MBELE BIDIMA Martin Ledoux	Chargé de Cours	En poste
18	MENGUE MENGUE David Joe	Chargé de Cours	En poste
19	NGUEFACK Bernard	Chargé de Cours	En poste
20	POLA DOUNDOU Emmanuel	Chargé de Cours	En poste
21	TAKAM SOH Patrice	Chargé de Cours	En poste
22	TCHANGANG Roger Duclos	Chargé de Cours	En poste
23	TCHOUNDJA Edgar Landry	Chargé de Cours	En poste
24	TETSADJIO TCHILEPECK M. E.	Chargée de Cours	En poste
25	TIAYA TSAGUE N. Anne-Marie	Chargée de Cours	En poste
26	DJIADEU NGAHA Michel	Assistant	En poste
27	MBIAKOP Hilaire George	Assistant	En poste
28	NIMPA PEFOUNKEU Romain	Assistant	En poste
29	TANG AHANDA Barnabé	Assistant	Directeur/MINTP
8- DÉPARTEMENT DE MICROBIOLOGIE (MIB) (13)			
1	ESSIA NGANG Jean Justin	Professeur	DRV/IMPM
2	ETOA François Xavier	Professeur	Chef de Département Recteur Université de Douala
3	NWAGA Dieudonné M.	Maître de Conférences	En poste
4	NYEGUE Maximilienne Ascension	Maître de Conférences	En poste
5	SADO KAMDEM Sylvain Leroy	Maître de Conférences	En poste
6	BOYOMO ONANA	Maître de Conférences	En poste
7	RIWOM Sara Honorine	Maître de Conférences	En poste
8	BODA Maurice	Chargé de Cours	En poste
9	BOUGNOM Blaise Pascal	Chargé de Cours	En poste
10	ENO Anna Arey	Chargée de Cours	En poste
11	ESSONO OBOUGOU Germain G.	Chargé de Cours	En poste
12	NJIKI BIKOÏ Jacky	Chargée de Cours	En poste
13	TCHIKOUA Roger	Chargé de Cours	En poste
9.DEPARTEMENT DE PYSIQUE(PHY)			
1	ESSIMBI ZOBO Bernard	Professeur	En poste
2	KOFANE Timoléon Crépin	Professeur	En poste
3	NDJAKA Jean Marie Bienvenu	Professeur	**Chef de Département**
4	NJOMO Donatien	Professeur	En poste
5	PEMHA Elkana	Professeur	En poste
6	TABOD Charles TABOD	Professeur	En poste
7	TCHAWOUA Clément	Professeur	En poste
8	WOAFO Paul	Professeur	En poste
9	EKOBENA FOUDA Henri Paul	Maître de Conférences	*Chef Division. UN*
10	NJANDJOCK NOUCK Philippe	Maître de Conférences	*Chef Serv. MINRESI*
11	BIYA MOTTO Frédéric	Maître de Conférences	**DG/Mekin**
12	BEN- BOLIE Germain Hubert	Maître de Conférences	CD/ENS/UN
13	DJUIDJE KENMOE épouse ALOYEM	Maître de Conférences	En poste
14	NANA NBENDJO Blaise	Maître de Conférences	En poste
15	NOUAYOU Robert	Maître de Conférences	En poste
16	SIEWE SIEWE Martin	Maître de Conférences	En poste
17	ZEKENG Serge Sylvain	Maître de Conférences	En poste
18	EYEBE FOUDA Jean sire	Maître de Conférences	En poste
19	FEWO Serge Ibraïd	Maître de Conférences	En poste

20	HONA Jacques	Maître de Conférences	En poste	
21	OUMAROU BOUBA	Maître de Conférences		
22	SAIDOU	Maître de Conférences	Sous Directeur/Minresi	
23	SIMO Elie	Maître de Conférences	En poste	
24	BODO Bernard	Chargé de Cours	En poste	
25	EDONGUE HERVAIS	Chargé de Cours	En poste	
26	FOUEDJIO David	Chargé de Cours	En poste	
27	MBANE BIOUELE	Chargé de Cours	En poste	
28	MBINACK Clément	Chargé de Cours	En poste	
29	MBONO SAMBA Yves Christian U.	Chargé de Cours	En poste	
30	NDOP Joseph	Chargé de Cours	En poste	
31	OBOUNOU Marcel	Chargé de Cours	Chef Service /Univ Douala	
32	TABI Conrad Bertrand	Chargé de Cours	En poste	
33	TCHOFFO Fidèle	Chargé de Cours	En poste	
34	VONDOU DerbetiniAppolinaire	Chargé de Cours	En poste	
35	WOULACHE Rosalie Laure	Chargée de Cours	En poste	
36	ABDOURAHIMI	Chargé de Cours	En poste	
37	ENYEGUE A NYAM épse BELINGA	Chargée de Cours	En poste	
38	WAKATA née BEYA Annie	Chargée de Cours	*Chef Serv. MINESUP*	
39	MVOGO ALAIN	Chargé de Cours	*Sous Directeur/MINESUP*	
40	CHAMANI Roméo	Assistant	En poste	
41	MLI JOELLE LARISSA	Assistante	*En poste*	
10- DÉPARTEMENT DE SCIENCES DE LA TERRE (ST) (42)				
1	NDJIGUI Paul Désiré	Professeur	**Chef de Département**	
2	BITOM Dieudonné	Professeur	*Doyen / FASA / UDs*	
3	NZENTI Jean-Paul	Professeur	En poste	
5	KAMGANG Pierre	Professeur	En poste	
6	MEDJO EKO Robert	Professeur	*Coseiller Technique/UYII*	
4	FOUATEU Rose épse YONGUE	Maître de Conférences	En poste	
7	NDAM NGOUPAYOU Jules-Remy	Maître de Conférences	En poste	
8	NGOS III Simon	Maître de Conférences	CD/Uma	
9	NJILAH Isaac KONFOR	Maître de Conférences	En poste	
10	NKOUMBOU Charles	Maître de Conférences	En poste	
11	TEMDJIM Robert	Maître de Conférences	En poste	
12	YENE ATANGANA Joseph Q.	Maître de Conférences	*Chef Div. /MINTP*	
13	ABOSSOLO née ANGUE Monique	Maître de Conférences	*Chef div. DAASR / FS*	
14	GHOGOMU Richard TANWI	Maître de Conférences	CD/UMa	
15	MOUNDI Amidou	Maître de Conférences	*Chef Div. MINIMDT*	
16	ONANA Vincent	Maître de Conférences	En poste	
17	TCHOUANKOUE Jean-Pierre	Maître de Conférences	En poste	
18	ZO'O ZAME Philémon	Maître de Conférences	*DG/ART*	
19	MOUNDI Amidou	Maître de Conférences	*Chef Div. MINIMDT*	
20	BEKOA ETIENNE	Chargé de Cours	*En poste*	
21	BISSO DIEUDONNE	Chargé de Cours	*Directeur/Projet Barrage Memve'ele*	
22	ESSONO Jean	Chargé de Cours	*En poste*	
23	EKOMANE EMILE	Chargé de Cours	*En pste*	
24	FUH Calistus Gentry	Chargée de cours	*Sec. D'Etat/MINMIDT*	
25	GANNO Sylvestre	Chargé de Cours	En poste	
26	LAMILEN BILLA Daniel	Chargé de Cours	En poste	

27	MBIDA YEM	Chargé de Cours	*En poste*
28	MINYEM Dieudonné-Lucien	Chargé de Cours	*CD/Uma*
29	MOUAFO Lucas	Chargé de Cours	En poste
30	NJOM Bernard de Lattre	Chargé de Cours	En poste
31	NGO BELNOUN Rose Noël	Chargée de Cours	En poste
32	NGO BIDJECK Louise Marie	Chargée de Cours	En poste
33	NGUETCHOUA Gabriel	Chargé de Cours	CEA/MINRESI
34	NYECK Bruno	Chargé de Cours	En poste
35	TCHAKOUNTE J. épse NOUMBEM	Chargée de Cours	*CT / MINRESI*
36	METANG Victor	Chargé de cours	En poste
37	NOMO NEGUE Emmanuel	Chargé de cours	En poste
38	TCHAPTCHET TCHATO De P.	Chargé de cours	En poste
39	TEHNA Nathanaël	Chargé de cours	En poste
40	TEMGA Jean Pierre	Chargé de cours	En poste
41	MBESSE CECILE OLIVE	Chargée de cours	En poste
42	ELISE SABABA	Chargé de cours	En poste
43	EYONG JOHN TAKEM	Assistant	En poste
44	ANABA ONANA Achille Basile	Assistant	En poste

Répartition chiffrée des Enseignants de la Faculté des Sciences de l'Université de Yaoundé I

NOMBRE D'ENSEIGNANTS					
DÉPARTEMENT	**Professeurs**	**Maîtres de Conférences**	**Chargés de Cours**	**Assistants**	**Total**
B.C.	5 (1)	10 (5)	21 (10)	3 (1)	**39 (17)**
B.P.A.	11 (1)	9 (3)	20 (8)	3 (5)	**43 (17)**
B.P.V.	4 (0)	9(2)	10 (2)	4 (4)	**27 (8)**
C.I.	10(1)	8(2)	16 (4)	0 (2)	**34 (9)**
C.O.	9 (0)	13 (3)	8 (2)	1 (0)	**31 (5)**
I.N.	3 (0)	1 (0)	8 (0)	12 (3)	**24 (3)**
M.A.	3 (0)	5 (0)	18 (1)	4 (0)	**30 (1)**
M.B.	2 (0)	5 (2)	6 (2)	0 (0)	**13 (4)**
P.H.	8 (0)	17 (0)	15 (2)	2 (1)	**42 (3)**
S.T.	5 (0)	15 (2)	23 (3)	2 (0)	**45 (5)**
Total	**60 (3)**	**92(19)**	**145 (34)**	**31(16)**	**328(72)**

Soit un total de **328 (**72**)** dont :

- Professeurs **60 (3**
- Maîtres de Conférences **92 (**19**)**
- Chargés de Cours **145 (34)**
- Assistants **31** (16**)**

( ) = Nombre de Femmes