David Oliveira Aparício

Pattern Discovery in Complex Networks using Parallelism

U. PORTO

FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2014

David Oliveira Aparício

Pattern Discovery in Complex Networks using Parallelism

U.PORTO

FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Ciência de Computadores

Advisors: Prof. Pedro Ribeiro and Prof. Fernando Silva

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2014

Para os meus pais e para as minhas irmãs

Acknowledgments

First and foremost, I want express my gratitude to my supervisors, Prof. Pedro Ribeiro and Prof. Fernando Silva, for their support, guidance and trust. I also want to thank them for the opportunities they have given me that have greatly contributed to my academic career.

I would like to thank the financial back up offered by FCT in project Sibila (Scientific Grant (NORTE-07-0124-FEDER-000059)). Before this grant I had a Scientific Initiation Grant from CRACS and I also want to thank them for it. I also thank Miguel Areias for providing me with tools that greatly helped me uncovering problems that would be very difficult to find without them. I also want to thank Hugo Ribeiro for answering all of my requests relative to the machines I used for development and testing purposes, both from CRACS and Sibila.

I want to thank my friends over these five years and particularly João Silva, for helping me decisively in the early times of this course and all throughout it, and Rafael Nunes and João Patrício for giving me many fun moments and interesting discussions. To Ana, for giving me focus during the six years that we spent together, helping and comforting me in too many situations to count.

Finally, I want to thank my parents for the love they have given me, for the crucial financial support and for enforcing the idea that I could always improve. And to my sisters that, despite being in another country, are always interested in offering me advice.

Abstract

Networks are used to represent systems in a multitude of fields, from biology and chemistry to telecommunication and transport networks. Networks have specific patterns that are characteristic to them and give important information relative to their structure and even to their function. Therefore, finding these relatively small patterns from large networks is an important data mining primitive. *Network motifs* and *graphlet based metrics* are examples of methodologies that look for small subgraphs in a large network.

Counting all occurrences of a set of subgraphs on a large network is, however, a *computationally hard task* closely related to the *graph isomorphism problem*, which has no known polynomial solution. A possible way to reduce the computing time necessary for this task is by avoiding isomorphism tests. The *g-trie* is a novel data structure that uses common sub-topologies between graphs to encapsulate isomorphism information, substantially reducing execution time when compared to past solutions.

To further speed up the execution of subgraph counting, we present parallel alternatives for both multicore and GPU architectures that build upon past g-trie algorithms. Both multicores and powerful GPUs are pervasive in today's personal computers, giving our work a broad scope.

We developed an efficient work sharing mechanism that performs dynamic load balancing of work units. We performed a thorough analysis of our algorithms' performance, using a large array of real-world networks from different fields, and obtained near-linear speedup for both multicore algorithms that we developed, showcasing the scalability of our approach. This work expands the applicability of subgraph counting algorithms for larger subgraph and network sizes without the obligatory access to a cluster.

Resumo

As *redes* são usadas para modelar sistemas em várias áreas, desde a biologia e a química até às redes de comunicação ou de transporte. As redes têm padrões específicos que as caracterizam e oferecem informação valiosa quanto à sua estrutura e mesmo da sua função. Encontrar esses padrões relativamente pequenos é, portanto, uma primitiva importante em *data mining. Network motifs* (padrões de rede) e métricas baseadas em *graphlets* são alguns exemplos de metodologias usadas com o propósito de encontrar pequenos subgrafos numa rede de grande dimensão.

Contar todas as ocorrências de um conjunto de subgrafos numa rede de grandes dimensões é, porém, uma *tarefa computacionalmente difícil*, fortemente ligada ao *problema do isomorfismo de subgrafos*, para o qual não se conhece nenhuma solução polinomial. Uma forma de reduzir o tempo computacional necessário para esta tarefa é evitando o recurso a testes de isomorfismo. A *g-trie* é uma estrutura de dados recentemente desenvolvida que usa subtopologias comuns entre grafos para encapsular informação relativa a isomorfismos, reduzindo substancialmente o tempo de execução quando comparado com soluções anteriores.

Com o fim de acelerar o processo de contagem de subgrafos, apresentamos alternativas paralelas para algoritmos que usam g-tries na sua base, quer para *multicores* como para GPUs. Tanto os *multicores* como os GPUs são ubíquos nos computadores pessoais de hoje em dia, dando ao nosso trabalho um vasto alcance.

Desenvolvemos um mecanismo de balanceamento de carga eficiente que divide o trabalho dinamicamente pelos recursos computacionais. Fizemos uma análise aprofundada do desempenho dos nossos algoritmos, usando para o efeito um conjunto vasto de redes reais provenientes de diversas áreas, e conseguimos escalar quase linearmente os dois algoritmos que desenvolvemos para *multicores*, provando a eficiência da nossa abordagem. Este trabalho possibilita o uso de algoritmos de contagem de subgrafos em redes de maior escala e para encontrar padrões maiores, sem ser necessário o acesso a um *cluster* dedicado.

Contents

List of Tables

List of Figures

List of Algorithms

Introduction 1

The chemical interactions between atoms give rise to molecules, some of them indispensable to the formation of the living organisms that inhabit the Earth. These living organisms constitute an intricate web of interactions between themselves and their environment. The planet they live in orbits around its star, as do seven other planets, and has itself enough gravitational pull to have a satellite of its own, with some of the other planets having much more than one. All this information, with much more detail, can nowadays be found on the Internet, a web of computers connected worldwide.

This kind of organization, with entities interacting and forming complex relations, can be found everywhere and our ability to recognize and build them is one of our strengths. *Networks* (or graphs) are an abstract model to represent these interactions.

The structure of a network is constituted by the connections between its entities. Studying it may give us new insights into the network function itself. In particular, finding patterns that appear throughout the network more times than would be expected on similar randomized networks is a method for analyzing the underlying design of networks from a wide variety of fields. These specific patterns are called network motifs and there have been significant research contributions since their introduction in 2002 [MSOI$^+$02].

One way to find patterns in a large network is to follow a bottom-up approach, starting from a single element and iteratively adding a new one from its neighborhood. However, this strategy gives rise to an exponentially large search space. Analyzing the structure of very large networks, comprising thousands or millions of nodes and edges, is not a trivial task and requires either a reduction of the search space or augmented computing power to be performed in a reasonable amount of time.

G-Tries are data structures specialized in efficiently storing graphs. They are akin to *prefix trees (tries)* but, instead of having each tree node represent (part of) a word, they represent (part of) a graph. How g-tries are created and how they can be used for subgraph frequency counting is detailed in Chapter 2. G-Tries are currently at the core of some of the fastest known methods for subgraph counting [RS10, PR13] and they achieve their heightened performance by narrowing the search space.

Before the early 2000s, personal computers consisted of a single core, with multicore machines being found almost exclusively on dedicated clusters available only at very high prices. Today, personal computers with up to 8 cores are ubiquitous and sold inexpensively. Powerful GPUs are also commonplace due to the ever-rising popularity of videogames and the pressure put on graphic processing companies to support games with increasingly realistic graphics and physics. With the progresses made in multicore and GPU hardware capabilities, problems previously unfeasible in a reasonable amount of time due to their exponential nature are now made possible. Strategies that manage to achieve almost linear speedups can reduce the computing time from weeks to days or hours, depending on the number of cores. With the commonplace 8 cores, programs that take 1 day sequentially can execute in 3 hours.

The aim of this thesis is to provide some of the fastest known ways to discover network patterns in large networks and, at the same time, make them widely available. For that purpose we implemented parallel strategies for both the multicore and GPU architectures, which are pervasive nowadays, using the g-trie data structure at the core of our algorithms.

1.1 Motivation

The network representation model is used in many different fields, including sciences like physics, chemistry and biology but also encompassing social networks, citation circles and transportation networks [dFCRTB07]. With their use being so widespread, a multitude of real-world data has been aggregated over the years, leading to the composition of several large data-sets. This extensive data collection has made it possible to study the structure of different types of networks, in a quest to find distinct topological features. Research has corroborated this view by finding that networks from different fields have specific patterns that distinguish them from random networks and from each other.

Network motifs are a possible way to discover patterns that would not likely be present in a random network. The concept was introduced by Milo et al. in [MSOI$^+$02] where they found that some small subnetworks appeared in the studied networks with a much higher frequency than it would be expected in similar randomized networks. Network motifs were then presented as the building blocks of complex networks and they have been used to study the structure of networks from various areas. They have been most notably studied in biological and biochemical networks [LBY$^+$04, SK04, KA05, PIL05, Kon08] and they have been extensively applied to protein-protein-interaction networks [YLSK$^+$04, AA04] and transcriptional regulatory networks [SOMMA02, LRR$^+$02, MKD$^+$04, LBY$^+$04, TZvO07]. Other areas where they have been studied include electronic circuits [ILK$^+$05] and software architecture networks [VS05].

Figure 1.1 displays a food-web network of the Maspalomas town in the Canary Islands [ABAU99]. It contains information from 18 living organisms and 3 kinds of carbon. We also show two network motifs, (a) and (b), and some of their occurrences in the food-web network. In the context of this network, (a) represents two organisms that share a food source but do not predate one another, while in (b) we have a circular food chain, with A eating C, C eating B and B eating A. Both subgraphs do appear in the network but their frequencies are quite different: (a) appears 82 times while (b) only 2. This appears to make sense for this kind of network since it seems more common in Nature for organisms to share food-sources than to be part of a circular food-chain. For other kinds of networks the frequencies could be very different.

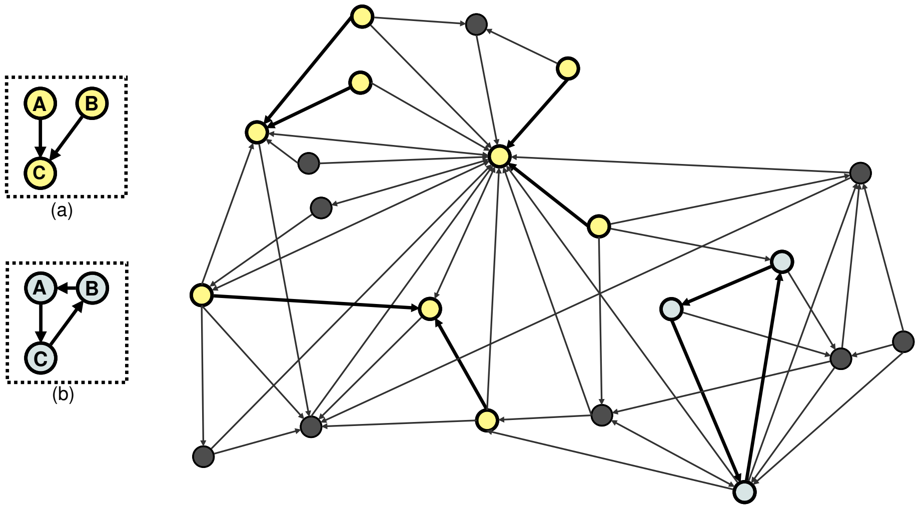


Figure 1.1: A food-web network highlighting some occurrences of two distinct network motifs.

The problem of finding subgraphs inside a larger graph is closely related to the *subgraph isomorphism problem* and is a known computationally hard (NP-Complete) problem. This means that, either by using larger networks or by querying bigger subgraphs, the time necessary for computation exponentially rises. It is therefore mandatory to reduce the enormous search space to a more manageable size.

A *g-trie* is a data-structure for storing and compressing a collection of graphs. They achieve that by building a tree where a path in it represents a way to arrive at a specific graph, with a new graph node being added in each tree descendent. Graphs may share a portion of a path if they have a common sub-topology. Looking at motifs (a) and (b) from Figure 1.1 we can see that they both share a connection from node A to node C. Therefore, by searching a portion of (a) we are, at the same time, searching the same portion of (b) and do not need to find them completely independently. If the number of graphs is sufficiently large, g-tries achieve a high compression of the search space. This makes them a good basis for subgraph counting algorithms since less independent isomorphism tests are needed.

The purpose of this work was to achieve the fastest known way to do subgraph frequency counting, also called *subgraph census* and we will use the two terms interchangeably. Our chosen method was to build parallel algorithms that use g-tries. We also intend to make our work available for any user, without the need of a dedicated cluster. With that in mind we targeted both the multicore and GPU architectures. The two of them are now commonplace, broadening the applicability of our work. By being able to find bigger patterns in a reasonable amount of time, researchers from many areas can more thoroughly analyze the structure of networks, possibly giving them a new insight into the field.

We conclude this initial chapter by presenting the graph terminology we use throughout this thesis and by more clearly defining what a *pattern* is in our context.

1.2 Graph Terminology and Subgraph Counting Problem Definition

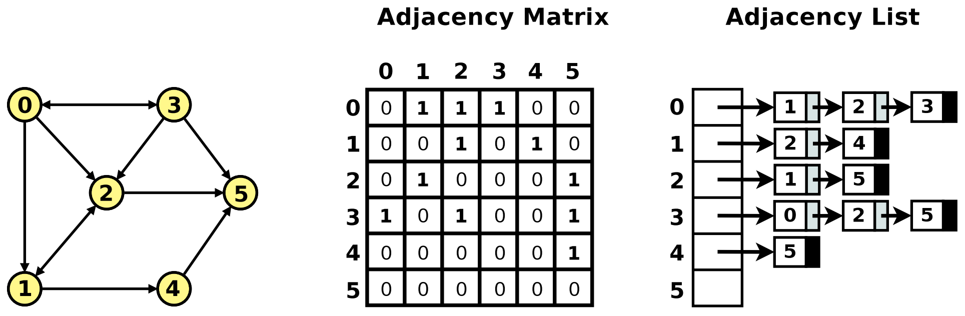
A network, or a *graph*, is comprised of a set $V(G)$ of *vertices* or *nodes* and a set $E(G)$ of *edges* or *connections*. The nodes represent entities and the edges relationships between them. The *size* of the graph is given by the number of vertices and is written as $|V(G)|$. Edges are represented as pairs of vertices of the form (a, b). In *directed* graphs the edges are *ordered pairs*, while in *undirected* graphs there is no order since the nodes are connected in both directions.

In undirected graphs, the *degree* of a vertex is the number of vertices v connected to vertex u. For directed graphs there are two types of degrees. The *indegree* of u is $|v|, (v, u) \in E(G)$. Similarly, the *outdegree* of u is $|v|, (u, v) \in E(G)$.

A graph is considered a *simple* graph if it has no self-loops (meaning it has no nodes connecting back to itself) and has no more than a single connection to any one node. The graphs we work with in this thesis are always simple graphs but g-tries can be adapted to support other kinds of graphs. The concept of *complex* networks is unrelated to this definition of simple graph. A network is said to be complex when it appears to have topological features that are neither purely random nor purely regular.

The set of vertices $v \in V(G)$ that have an edge (v, u) constitute the *neighborhood*, $N(u)$, of $u \in V(G)$. To easily recognize vertices they are labeled from 0 to $|V(G)|$ - 1. The comparison $v < u$ translates to v having a lower label index than u. Two graphs are said to be *isomorphic* if we obtain one from another just by changing the node labels, without affecting the connections between them.

Two possible representations of a graph are the adjacency matrix and adjacency list formats. In the matrix representation, denoted by G_M, an edge (u, v) is represented by $G_M[u][v]$. The matrix has an entry for every possible edge, if $(u, v) \in E(G)$, $G_M[u][v]$ has value 1, and 0 otherwise. In an adjacency list, each row represents the connections of a particular vertex. This means that not every possible edge is stored, saving storage space, but makes the process of checking an edge connection more computationally expensive (which could be done using binary search, taking logarithmic instead of the constant time required when using an adjacency matrix). In Figure 1.2 a graph and its respective representations in adjacency matrix and adjacency list are presented. Generally, the matrix representation is well-suited for dense graphs whereas adjacency lists are most often used for sparse graphs.



	0	1	2	3	4	5
0	0	1	1	1	0	0
1	0	0	1	0	1	0
2	0	1	0	0	0	1
3	1	0	1	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	0

Figure 1.2: A graph and its representation in the adjacency matrix and adjacency list formats.

A *subgraph* G_k of G is a graph of size k where $V(G_k) \subseteq V(G)$ and $E(G_k) \subseteq E(G)$. A subgraph is *induced* if $\forall u, v \in V(G), (u, v) \in E(G_k)$ if and only if $(u, v) \in E(G)$. A *match* or *occurrence* happens when G has a set of nodes that induce some G_k. Two matches m_1 and m_2 are considered distinct if they have at least one different vertex.

The *frequency* of G_k in G is the number of occurrences of G_k in G. Figure 1.3 gives an example of the frequency concept explained before.

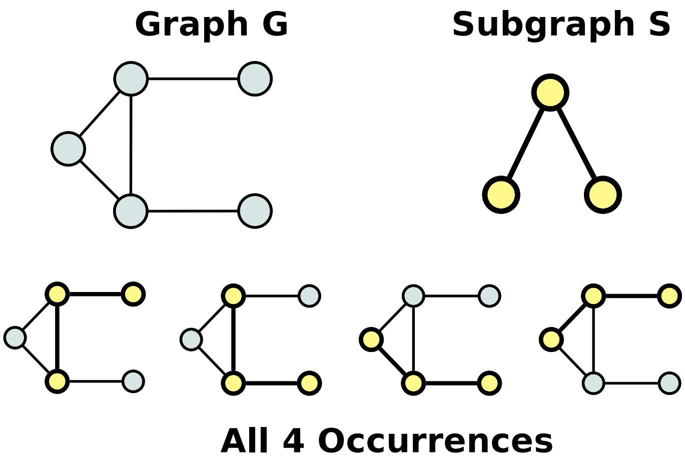


Figure 1.3: Occurrences of a subgraph S in a larger graph G.

The purpose of this work is to find the frequency of all possible subgraphs of a given size in a single large graph. A formal definition is given in Definition 1.1.

Definition 1.1 (**Subgraph Census Problem**)**.** *Given a set of all subgraphs of size k and a graph G, determine the exact frequency of all induced occurrences of the subgraphs in G. Two occurrences are considered different if they have at least one node or edge that they do not share. Other nodes and edges can overlap.*

1.3 Pattern Definition

There are numerous ways a *pattern* can be defined in the context of graphs. Next we introduce fields of study strongly related to the *subgraph census problem* that find and classify patterns in different fashions.

1.3.1 Network Motifs

The concept of network motifs was first presented in [MSOI$^+$02] as small subgraphs that appeared in a network with a higher frequency than in similar randomized networks. The authors introduced them as basic structural elements that could effectively characterize networks into classes. They were also confident that these patterns could not only define the structure of the networks, but also give hints to its function.

To study network patterns using motifs it is required to build a set of random networks that are similar to the original. The motifs are then first searched in the original network and afterwards on the set of similar random networks. How these random networks are considered similar to the original and how we calculate if a pattern is over-represented may differ according to our goal, but here we will describe the standard and most common way to accomplish these tasks.

An undirected random network similar to the original is generated by creating a new network with the same amount of vertices and edges. The set of (u, v) is randomized while keeping the degree (or *indegree* and *outdegree*, in the case of directed networks) of each vertex. An example of a set of similar random networks is shown in Figure 1.4.

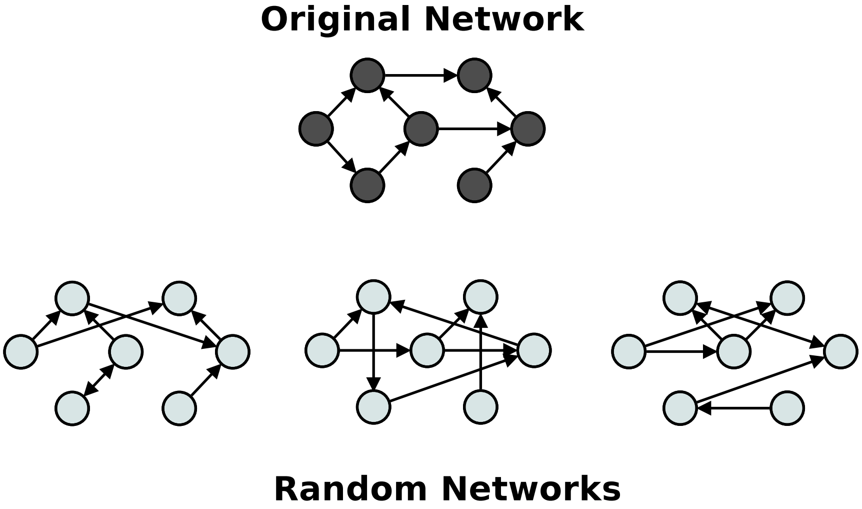


Figure 1.4: A network and a set of 3 similar random networks, keeping degree sequence.

The majority of network motif related papers require the following properties to hold:

- **Over-representation:** The probability that the motif appears more times in a similar random network should be less than some pre-calculated probability P.
- **Minimum frequency:** For a motif to be significant it should have a frequency higher than some set threshold. It is left to the user the choice of a proper threshold, giving flexibility to the definition.
- **Minimum deviation:** Finally, the motif needs to appear with a substantially higher frequency in the original network than its average frequency in the random networks. Once again, what is considered *significant* depends on the network being studied and what the researcher pretends to achieve.

Network motifs can be applied to either directed or undirected networks in the same manner. The concept has been extended to colored networks, with *colored motifs* having to check not only the connections of the vertices but also their color [LFS06, FFHV07, RS14a]. *Anti-motifs* are a variant of the motif concept with the purpose of finding under-represented patterns that may be meaningful depending on the application [MIK$^+$04, JJCL06, BGP07]. Another example of an extension to the original network motifs adds in the *weight* of the connections [OSKK05, CRS12]. Finally, the concept of *trend motifs* was introduced to account for dynamic networks that change over time and have recurring subgraphs with similar dynamics over a period of time [JMA07].

In their seminal paper, Milo et al. [MSOI$^+$02] proposed a brute-force algorithm that paved the way for future improvements. Some of the algorithms for motif discovery include `mfinder` [KIMA04], `ESU` [WR06], `Kavosh` [KAE$^+$09], `gtrieScanner` [RS10] and `FaSE` [PR13]. Currently, `gtrieScanner` and `FaSE` are the two fastest algorithms for subgraph census that the authors know of, performing, generally, one or two orders of magnitude faster than competing algorithms.

1.3.2 Frequent Subgraph Mining

Network motifs are used to find the set of subgraphs that are overrepresented in a large single graph. The concept of *frequent subgraph mining* (FSM) is broader and usually divided into two problem formulations: (i) *graph transaction based FSM* and (ii) *single graph based FSM*. In *graph transaction based FSM* the input consists of a set of graphs, called *transactions*, a term borrowed from *association rule mining* [AS94], another Data Mining field.

For a subgraph to be considered frequent, its frequency has to be greater than some predefined threshold. The frequency of a subgraph in FSM is commonly called *support*. The support of a subgraph may be computed either in a *transaction-based* or *occurrence-based* fashion. Transaction-based means that the support of a subgraph is equal to the number of transaction in which it appears divided by the total amount of graphs. On the other hand, in occurrence-based counting the support of the graph is the total number of times it appears, either in the single graph or in the set of transactions. In single graph based FSM the only possible method for counting is occurrence-based while graph transaction based FSM most often uses transaction based counting [JCZ13], thus, two definitions are given next.

Definition 1.2 (**Graph Transaction Based FSM**). *Given a collection of* n *transactions* $\mathcal{G} = \{G_1, G_2, ..., G_n\}$*, find all frequent subgraphs in* $\mathcal{G}$*. The support of a subgraph* S *is* $sup_{\mathcal{G}}(S) = |\{G_i | S \subseteq G_i\}|/n$*.* S *is considered frequent if* $sup_s \geq \delta$*, with* δ *as a predefined value where* $0 < \delta \leq 1$*.*

Definition 1.3 (**Single Graph FSM**). *Given a single graph* G*, find all its frequent subgraphs. The support* sup_s *of a subgraph* S *is* $G(S) = |\{G_i | S \subseteq G_i\}|$*.* S *is considered frequent if its support* $sup_s \geq \delta$*, where* δ *is a predefined value bigger than 0.*

Algorithms for FSM typically start by generating the possible candidates and then doing the census on the set of graphs. Candidate generation can be done using either breadth-first or depth-first search. The *downward closure property* used in *frequent itemset mining* is also prevalent in FSM algorithms. Applied to graphs, what the property essentially says is that if a graph is not frequent, a supergraph containing it will also not be frequent. Similarly, if a graph is frequent, its subgraphs are assuredly frequent.

Numerous algorithms for FSM exist in the literature. Some of the most well known include `MoFa` [BB02], `gSpan` [YH02], `FFSM` [HWP03] and `Gaston` [NK04]. There have been experimental studies to compare the performance of these and some other algorithms from the field and the general consensus is that there is no clear best algorithm for all cases, the performance depends on the type of network being studied, the size of the graphs and memory constraints [WMFP05, KRSA11].

1.3.3 Graphlet Degree Distributions

A similar concept to network motifs is *graphlets* [PCJ04]. They are also small graphs that can be seen as building blocks of a network, the main difference being that random networks are not used to verify their over-representation. Figure 1.5 shows the set of the 29 graphlets of sizes 3 to 5, taken from [PCJ04].

Basically, a set of graphlets is chosen and their frequency is computed on a set of networks. The similarity of two networks is calculated as the difference between their graphlet frequencies.

To calculate the relative frequency of a graphlet i we simply divide its frequency by the frequency of all n graphlets combined. Since graphs can immensely differ in number of nodes and edges, the negative logarithm of that number is used, as displayed in Equation 1.1. The relative graphlet frequency distance between two graphs G_1 and G_2 can then be calculated as shown in Equation 1.2.

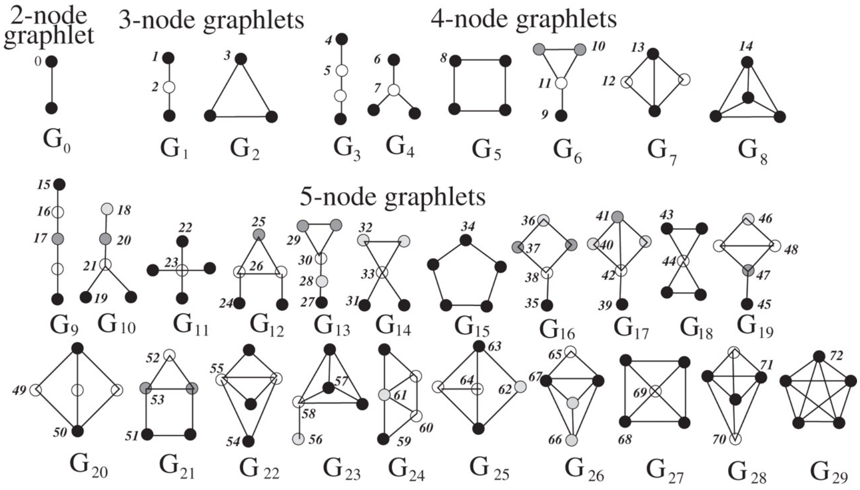


Figure 1.5: The set of all possible graphlets of sizes 3 to 5.

$$Rfreq_i(G) = -log(\frac{Freq_i}{\sum_{i=1}^{n} Freq_i(G)}) \tag{1.1}$$

$$Dist(G_1, G_2) = \sum_{i=1}^{n} |Rfreq_i(G_1) - Rfreq_i(G_2)| \tag{1.2}$$

Some applications using graphlets include the concept of orbits that are, essentially, the different kinds of nodes that the graphlets have. The set shown in Figure 1.5 has 72 different orbits. Nodes from a graphlet are counted as the same orbit if they have the same type of connections. For example, graphlet $G2$ has 3 nodes all pertaining to the same orbit while $G6$ has 4 nodes and 3 different orbits. Using this method we can not only count the frequency of each subgraph (or graphlet) but also how many times each node appears in each position of a subgraph (or orbit).

Graphlets have been used to study biological [Prž07, MP08], biochemical [PCJ06] and social networks [JHK12].

1.4 Thesis Outline

This thesis is structured in six major chapters. A brief description for each one is provided below.

Chapter 1 - Introduction. Offers an overall view of the problem being studied in this thesis as well as the motivation behind it. Elaborates on the problem definition and the graph terminology adopted throughout this work. Also discusses related problems and their importance. Additionally presents the thesis organization and mentions the scientific work published.

Chapter 2 - The G-Trie Data Structure. An explanation of how g-tries are created and how they are used for sequential subgraph census is provided. Here we also discuss two algorithms that are built on top of the g-trie data-structure, their similarities and differences. We end by discussing the opportunities for parallelism offered by g-tries.

Chapter 3 - Parallel Subgraph Census for Multicores. We justify the option to apply parallel computing to our particular problem and provide a overview of the related work. Our two parallel algorithms for multicores and the general sharing mechanism that we developed are detailed.

Chapter 4 - Parallel Subgraph Census using GPUs. Discusses the GPU architectural model and the problems it inflicts on graph traversal algorithms. Also presents an initial algorithm for GPUs using `CUDA`.

Chapter 5 - Performance Evaluation. We access the scalability of our multicore implementations by doing a thorough study using a dozen of large scale networks from different fields. We also evaluate our GPU algorithm and identify the problems with its efficiency.

Chapter 6 - Concluding Remarks. Concludes the thesis with the progresses achieved and gives directions for future work.

1.5 Bibliographic Note

Our pattern discovery parallel strategy and implementation for multicores resulted in two papers that were accepted for publication in peer-reviewed conferences. In the papers we used the g-trie data structure and achieved almost linear speedup for a set of wide-range large scale networks, both for `gtrieScanner` [ARS14] and `FaSE` [APR14].

The G-Trie Data Structure 2

In this chapter we present the novel data structure that we use at the core of our algorithms, the g-trie. We discuss how g-tries are created, how they can be used for subgraph census and how their structure offers opportunities for parallelism. For this purpose we detail two sequential algorithms that use g-tries to encapsulate isomorphism information.

2.1 Basis and Motivation

G-Tries were primarily designed for finding network motifs [RS10]. Algorithms for network motif discovery can, traditionally, be separated in two distinct approaches: *network-centric* and *subgraph-centric*. Network-centric methods start by finding all occurrences of k connected nodes in the network and then performing isomorphism tests to determine which subgraph type each occurrence belongs to. On the other hand, subgraph-centric methods pre-compute a list of all subgraphs to be searched and then find the occurrences of each type, one at a time, in the network. `ESU` [WR06], `Kavosh` [KAE$^+$09] and `FaSE` [PR13] are example of network-centric methods while Grachow and Kellis' algorithm [GK07] is subgraph-centric.

Typical g-tries, as used in [RSL10b], stand conceptually in the middle as *set-centric*: not just one subgraph is searched at a time but neither are, necessarily, all subgraphs of a certain size k. Compared to past competing sequential algorithms for motif discovery they were shown to perform one or two orders of magnitude faster. Their enhanced performance comes from the way g-tries heavily constrain the search space. This is done by identifying common structures between the different subgraphs that are searched in the larger network.

The concept is similar to that of a *prefix tree* (or trie) [Fre60]. A prefix tree of a set of words identifies *sub-words* that start with the same letters and represent them as the tree nodes. In this fashion, a full word is a path in the trie and the words in that path share a common sub-topology (or a *prefix* in this case). Figure 2.1 is an example of a prefix tree built from the words in the phrase *"The sixth sick sheikh's sixth sheep's sick"*. White letters on a black background are the letters being added at that trie node while black letters on a white background are words from the ancestor nodes. Prefix trees can thus be used to compress information.

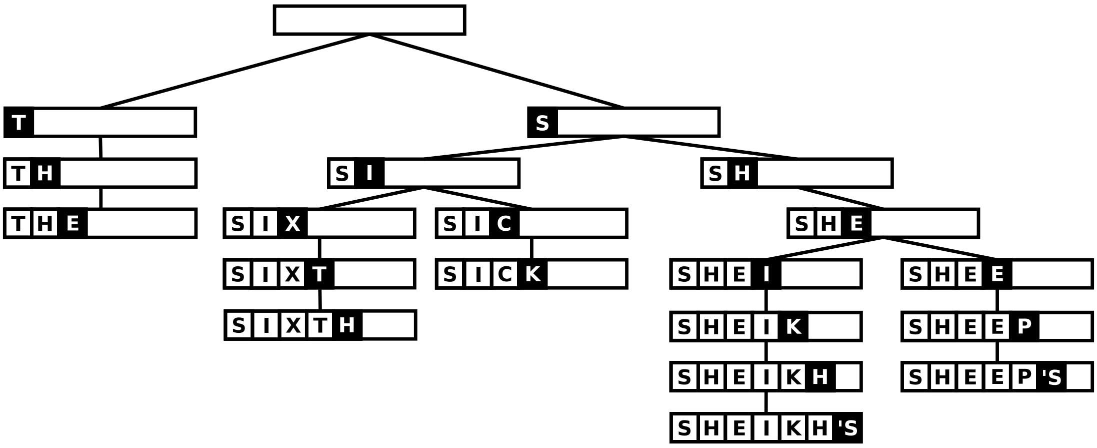


Figure 2.1: A prefix-tree (trie) of 5 words.

The same thinking can be applied to graphs. In Figure 2.2 we show three graphs containing the same 3-node sub-topology. Whereas tries use nodes to represent words, g-tries use nodes to represent graphs instead, hence the name *g(raph)-tries*.

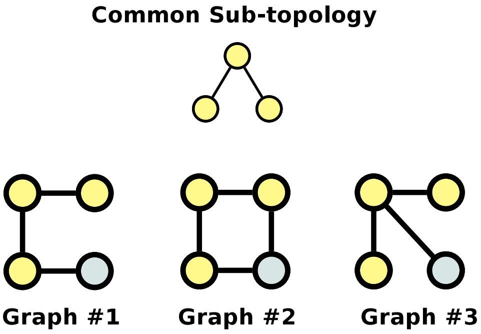


Figure 2.2: Common sub-topology between three non-isomorphic graphs.

An example g-trie built to contain all undirected graphs of size 4 is presented in Figure 2.3. To avoid confusion, henceforth we will use *nodes* to mean nodes of the g-trie and *vertices* as vertices of the network or vertices of the g-trie nodes. The vertices in clear color are the vertices already in the parent g-trie node and the node in black is the new one being added. The edges between the g-trie nodes mean that the children nodes share the common sub-topology of the parent node. The conditions of the form $X < Z$ translate to *"Vertex Z needs to have an higher index than vertex X"* and are used to deal with isomorphisms.

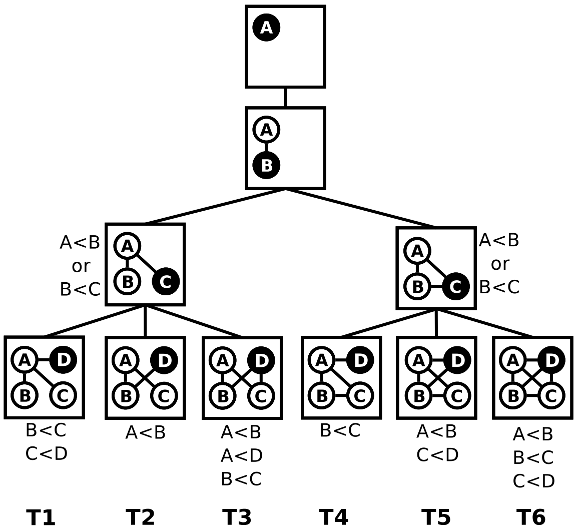


Figure 2.3: A g-trie of all size-4 undirected graphs.

2.2 Subgraph Counting using G-Tries

As stated in Definition 1.1, our problem is to find all subgraphs of size k in a large network G. We will now explain how g-tries are built and can efficiently be used for that purpose. With that objective we detail two algorithms, `gtrieScanner` and `FaSE`, that rely on g-tries to heavily constrain the induced search space.

2.2.1 G-Trie Creation

G-Tries can either be fully constructed before the subgraph census begins or be built on-the-fly. `gtrieScanner` and `FaSE` diverge on this option, with the first having a g-trie built before computation starts and the latter building it during execution. For `gtrieScanner`, a g-trie could also be stored in a file and loaded to memory prior to the computation.

Before creating the g-trie, it is necessary to establish a *canonical representation* since there exist many possible ways, by modifying the vertex labels, to represent the same graph. These distinct representations give origin to different g-tries because the order of the nodes influences how fast a common sub-topology is found. For instance, using a representation that creates a g-trie node with a new vertex that has no connections to previous vertices (a list with only zeros, in the adjacency matrix format) will not be very effective to achieve a good compression since, certainly, that vertice will have a future connection that could share a common sub-topology. Therefore, some care has to be taken in order to achieve a canonical representation that creates the least possible amount of nodes in the g-trie.

Regardless of when the g-trie is created, the creation process itself is similar. In Figure 2.4 we show an empty g-trie being populated with four size-4 graphs. Dotted nodes are nodes that the g-trie had to build to accommodate the new graph. Nodes with a grey background represent the path in the g-trie leading to the newly inserted graph. Notice that, while in the first step all 4 g-trie nodes had to be built, in the second step only a new node was created due to the shared topology between the two graphs. Without the g-trie, these four graphs would need a representation of 16 nodes (4 for each graph) but, using g-tries, only 8 nodes were necessary, achieving a compression rate of 50% for this small example. For larger collections of graphs the compression rate gets increasingly more significant. In Table 2.1 we display the compression rates for all undirected graphs of different sizes and we can see that, as the size gets bigger, the compression rate gets close to 90%. G-Tries heavily constrain the search by having different subgraphs in the same path being searched at the same time, without the need to do the full computation for all distinct subgraphs.

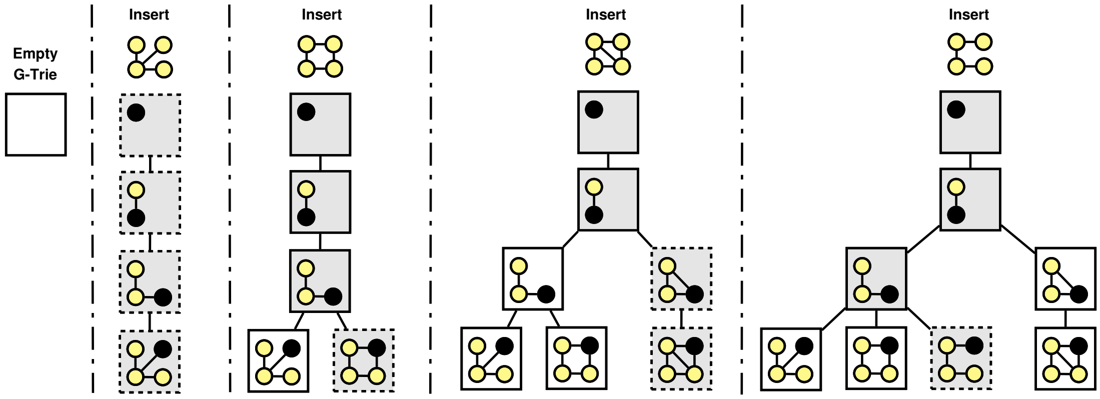


Figure 2.4: A g-trie being populated with four size-4 graphs.

Subgraph Size	#Graphs in the G-Trie	Compression
3	2	33%
4	6	58%
5	21	70%
6	112	78%
7	853	82%
8	11,117	85%
9	261,080	87%

Table 2.1: Compression rates of the search space achieved by using g-tries.

2.2.2 gtrieScanner

After a g-trie is created it can be used to count subgraph frequencies. The core idea of the algorithm is to search for a set of vertices V_{used} that match a path in the g-trie, thus corresponding to an occurrence of the subgraph represented by that path. To illustrate how this is done, we show two occurrences with a shared common sub-path being mapped into two distinct g-trie subgraphs in Figure 2.5. For these two occurrences, V_{used} is $\{3, 7, 2, 6\}$ and $\{3, 7, 4, 5\}$, respectively. The first two vertices, corresponding to $\{3, 7\}$, are common to the two subgraphs.

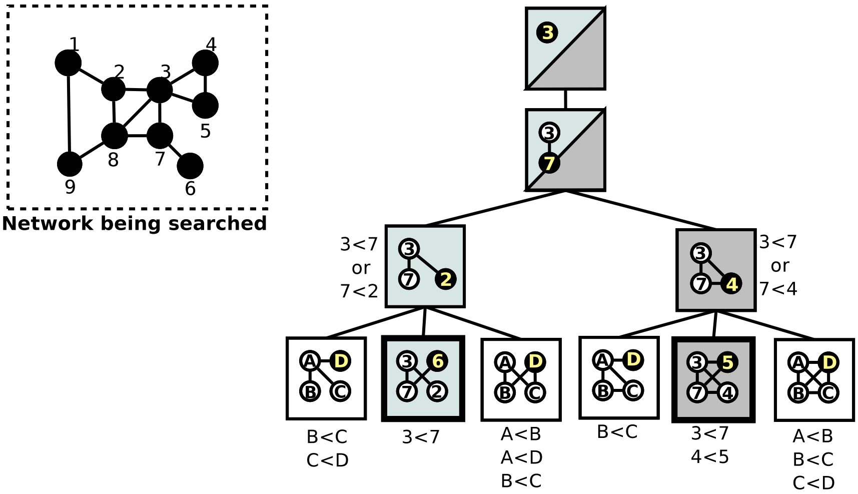


Figure 2.5: Two occurrences of distinct subgraphs, mapped to a g-trie.

Algorithm 2.1 depicts the pseudo-code of `gtrieScanner` used to compute the frequency of the subgraphs stored in a g-trie T in graph G. In the beginning, all vertices from G are possible candidates for the initial g-trie root node (lines 2 to 4) since there are no connections in the g-trie that they must match. Note that this is only true since we are not considering self-loops, in which case two root nodes would be necessary, one with the self-loop and one without it. The algorithm proceeds by looking for the set of vertices that fully match with the current g-trie node (line 6) and traverses that set. If it arrives at a leaf, it has found an occurrence of a subgraph and increments its respective frequency (line 9). If not, the algorithm continues recursively to the other possible g-trie descendants until it reaches a leaf. Function `matchingVertices()` gives some detail on how matches for the current g-trie node are efficiently found. We start by, at the current partial match, looking for the vertices that are connected to the vertex being added (line 14). To clarify this point we will make use of Figure 2.3 where, for subgraph $T1$, we can see that the only vertex connected to D is A, so it would be the only valid option for V_{conn}, whereas in $T6$, A, B and C are all connected to D, so they are all put in V_{conn}. We then look for the vertex m from V_{conn} with the smallest number of neighbors in the input network (line 15). The candidates for the new position are then the vertices that are neighbors of m, have the exact set of needed connections with the already matched vertices and also respect the symmetry breaking conditions stored in the g-trie node (lines 16 to 18).

Algorithm 2.1 `gtrieScanner`: Algorithm for computing the frequency of subgraphs stored in a g-trie T in graph G.

```
 1: procedure COUNTALL($T, G$)
 2:     for all vertex $v$ of $G$ do
 3:         for all child $c$ of $T.root$ do
 4:             COUNT($c$, $\{v\}$)
 5: procedure COUNT($T, V_{used}$)
 6:     $V \leftarrow$ MATCHINGVERTICES($T, V_{used}$)
 7:     for all vertex $v$ of $V$ do
 8:         if $T.isLeaf$ then
 9:             $T.frequency$++
10:         else
11:             for all child $c$ of $T$ do
12:                 COUNT($c, V_{used} \cup \{v\}$)
13: function MATCHINGVERTICES($T, V_{used}$)
14:     $V_{conn} \leftarrow$ vertices connected to the vertex being added
15:     $m$      $\leftarrow$ vertex of $V_{conn}$ with smallest neighborhood
16:     $V_{cand} \leftarrow$ neighbors of $m$ that respect both
17:               connections to ancestors and
18:               symmetry breaking conditions
19:     return $V_{cand}$
```

To further illustrate how g-tries work, we will now exemplify how one occurrence is found using Figures 2.3 and 2.5. We use the notation *(X, k)* to denote that vertex k is matched to X in the g-trie node. Take for instance the occurrence $\{2, 3, 7, 6\}$ of type $T2$ subgraph. Looking at the respective g-trie leaf, we can see that the only path leading to this occurrence will be $(A, 3) \rightarrow (B, 7) \rightarrow (C, 2) \rightarrow (D, 6)$. A path like $(A, 2) \rightarrow (B, 3) \rightarrow (C, 7) \rightarrow (D, 6)$ could not happen because if we added $(C, 7)$ there would be no matching g-trie node regarding the connections. A path like $(A, 7) \rightarrow (B, 3) \rightarrow (C, 6) \rightarrow (D, 2)$ could not happen either because, even if that corresponded to valid connections, it would break symmetry conditions. In particular, $T2$ imposes the condition $A < B$ which is false since 7 is not smaller than 3. These two simple mechanisms (verifying connections and symmetry conditions) form the basis of how a g-trie is able to highly constrain and limit the candidates it is checking and, at the same time, guarantee that each occurrence is found only once.

2.2.3 FaSE

gtrieScanner offers a set-centric approach to subgraph counting by giving the user the ability to search only the set of graphs that the user wishes to find, and not necessarily just one graph at a time nor all possible graphs. A downside of this strategy is that the g-trie created may have graphs that do not appear in the queried network, resulting in wasted storage space. FaSE avoids this problem by using g-tries differently: instead of building a complete g-trie before the subgraph census, the g-trie is constructed on-the-fly with each g-trie node being added only when it appears on the network.

Like previous algorithms such as ESU or Kavosh, FaSE follows a network-centric paradigm. However, contrarily to them, FaSE does not withhold the isomorphism tests until the end of the enumeration. Instead, it checks if two subgraphs belong to the same intermediate class during the actual enumeration process. Thus, a single isomorphism test per intermediate class is needed, contrasting with previous methods that required one per subgraph occurrence. This leads to a major speed up when compared to past algorithms, since the number of intermediate classes will assuredly be much smaller than the total number of subgraph occurrences, which is corroborated empirically. Comparing FaSE's generated g-trie with the g-trie from Figure 2.3, a major difference is that symmetry conditions do not exist. Thus, in FaSE, it is possible for isomorphic graphs to exist in the same g-trie, as can be seen in Figure 2.6. To deal with this, FaSE instead labels each g-trie node and performs isomorphic tests at the leaves to ensure that, despite following a different path in the g-trie, isomorphic graphs are counted as the same subgraph class [PR13].

In practice the algorithm performs two main intertwined tasks: enumerating subgraphs and storing isomorphism information in a g-trie. The enumeration process simply iterates through each subgraph occurrence, similarly to previous network-centric methods. At the same time, a tree is used to encapsulate the topological features of the enumerated subgraphs. It does so by generating a new label, which represents further information from each newly added vertex and uses it to describe an edge in a tree. This effectively partitions the set of subgraphs into intermediate classes. The entire process is summarized in Algorithm 2.2.

FaSE essentially works by enumerating all size k subgraphs only once. It does so by keeping two ordered sets of vertices: V_s and V_{ext}. The former represents the partial subgraph that is currently being enumerated as a set of connected vertices. The latter represents the set of vertices that can be added to V_s as a valid extension. To start the counting process, the algorithm initializes an empty g-trie (line 2). Each vertex v in

Algorithm 2.2 `FaSE`: Algorithm for computing the frequency of all subgraphs of size k in graph G

```
 1: procedure FaSE($G, k$)
 2:     initGTrie($T$)
 3:     for all vertex $v$ of $G$ do
 4:         enumerate($\{v\}, \{u \in N(v) : u > v\}, T.root$)
 5:     for all $l$ in $T.leaves()$ do
 6:         $frequency$[canonicalLabel($l.Graph$)] += $l.count$
 7: procedure enumerate($V_s, V_{ext}, current$)
 8:     if $|V_s| = k$ then
 9:         $current.count$++
10:     else
11:         for all vertex $v$ in $V_{ext}$ do
12:             $V'_{ext} \leftarrow V_{ext} \cup \{u \in N_{exc}(v, V_s) : u > V_s[0]\}$
13:             $V'_s \leftarrow V_s \cup \{v\}$
14:             $current' \leftarrow current.Child$(NewLabel($V_s$))
15:             Enumerate($V'_s, V'_{ext}, current'$)
```

the network is used to set $V_s = \{v\}$ and $V_{ext} = N(v)$, where $N(v)$ are the neighbors of v (lines 3 and 4). Then, one element u of V_{ext} is removed at a time, and a recursive call is made adding u to V_s and each element in $N_{exc}(u, V_s)$ with label greater than $V_s[0]$ to V_{ext} (lines 11 to 15). $N_{exc}(u, V_s)$ are the exclusive neighbors, that is, the neighbors of u that are not neighbors of V_s. This, along with the condition $u > V_s[0]$, ensures that there is no subgraph enumerated twice. When the size of V_s reaches k the algorithm has found a new occurrence of a size k subgraph (lines 8 and 9).

The enumeration step is wrapped by a g-trie that stores information of the subgraphs being enumerated in order to divide them into intermediate classes, one class in each of the g-trie nodes. When adding a new vertex to the current subgraph, a label is generated describing its relation to the previously added vertices (line 14). This label will govern the edges in the tree, that is, each edge is represented by a label generated by a vertex addition.

This label creation process is required to run in polynomial time since, otherwise, it could even become computationally heavier to use than simply doing the isomorphism test (NP-C). Thus there is a trade-off between time spent creating the label and time spent enumerating and running isomorphism tests on subgraphs. For this work we use an *adjacency list labeling*, which generates a label corresponding to an ordered list of at most $k-1$ integers where each value i $(0 < i < k)$ is present if there is a connection from the new vertex to the i-th added vertex.

Figure 2.6 summarizes the whole algorithm. The tree on the left represents the implicit recursion tree that `FaSE` creates during enumeration. Note that it is naturally skewed towards the left. This is an important fact that justifies, as we will see later, the need to redistribute work in the parallel version of the algorithm. The induced g-trie on the right is a visual representation of the actual g-trie that `FaSE` creates.

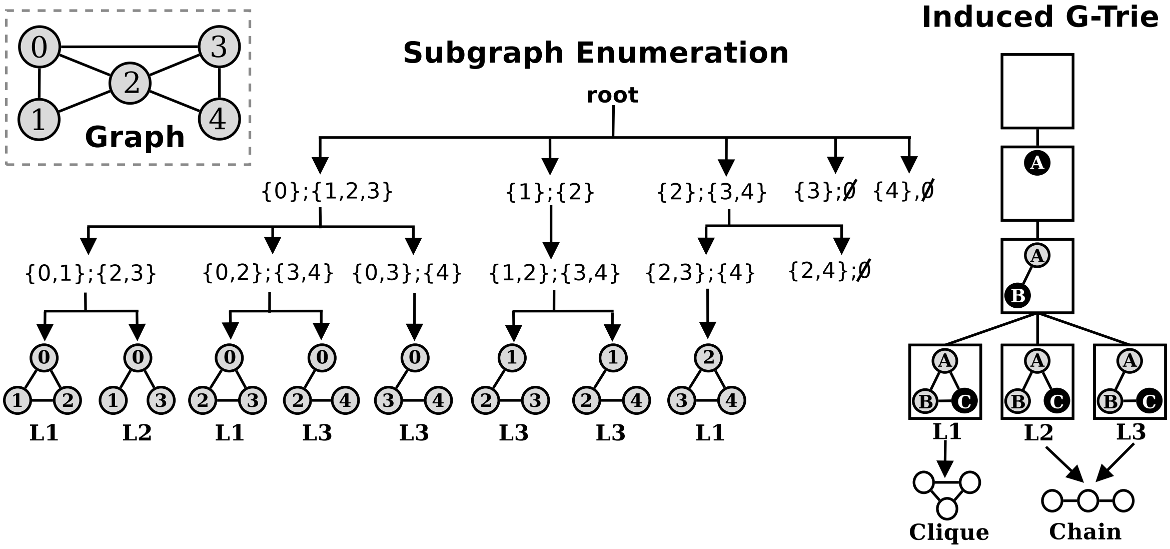


Figure 2.6: Summary of the enumeration and encapsulation steps of `FaSE`.

2.3 Opportunities for Parallelism

One of the most important aspects of both g-trie sequential algorithms is that they generate completely independent search tree branches. The order by which they are explored is also irrelevant for the frequency computation. In fact, looking at Figure 2.1, we observe that each call to `count`(T, V_{used}) produces a new different branch and the same is true for `FaSE`. Knowing the g-trie node T and the already matched vertices V_{used} (or V_s and V_{ext} for `FaSE`) is sufficient for continuing the search from that point. Each of these calls can thus be thought of as a *work unit*. This independence between work units makes work division a less cumbersome task.

For `gtrieScanner`, another factor that facilitates the parallelization is the fact that neither the original network nor the g-trie are changed during computation. This removes the need to guarantee safe writes to the centralized structures. As discussed earlier, in `FaSE` the g-trie undergoes changes during execution and special care must be taken to ensure consistency.

2.4 Summary

In this chapter we gave a general overview of the g-trie data structure and how it can be used for the general subgraph counting problem that we tackle in this work. We detailed two efficient sequential algorithms that use g-tries to store subgraph isomorphism information. We also put forward some key points of g-tries that make them a good basis for parallelization.

Parallel Subgraph Census For Multicores 3

The purpose of this chapter is to detail our two parallel algorithms that use g-tries to perform subgraph counting. We parallelized both `gtrieScanner` [RS10][1] and `FaSE` [PR13]. Despite the differences highlighted in the previous chapter, the two algorithms are functionally similar and what is discussed in this chapter applies to both, unless stated otherwise. Namely, the workload balancing mechanism suffers only minor changes between them and is general enough to be applicable to other subgraph census algorithms.

We will first discuss why parallelization is an appropriate solution to improve our algorithms' execution time and why it can be successfully applied to subgraph census, despite proving to be a challenging task.

3.1 Motivation

Improving the execution time of subgraph counting can have a broad impact. For instance, even increasing by just one node the size of the subgraphs being searched may lead to the discovery of new patterns, providing a new insight into the network. Since networks are present virtually in every field of study, from biology to chemistry and computer science, this can impact a multitude of areas.

One possible way to improve an algorithm's execution time is by using parallel computing. The idea is to split the work between CPUs, or the cores inside the same CPU, effectively reducing the time necessary to perform the task when compared with using just one unit to compute the complete task. At this point the distinction between computers and cores is not relevant and we will use *cores* as the general term for both.

Speedup S_i is calculated by dividing the time that a single core took to execute some task T_1 by the time T_i that i cores expended in the same task, as shown in Equation 3.1.

[1] The code for both the sequential algorithm and our parallel version can be found at the following URL: `http://www.dcc.fc.up.pt/gtries/`

$$S_i = \frac{T_1}{T_i} \tag{3.1}$$

Linear speedup is obtained when, by using k cores, we reduce the computing time by a factor of k. If a program with linear speedup takes 1 day to run *sequentially* (meaning in a single core), it would only take 1 hour using 24 cores.

However, achieving linear speedup is usually not a trivial task and depends on many factors. Some programs are inherently sequential, meaning that they can not be efficiently parallelized. On the other hand, *generating all possible permutations of a given size* is an example of an intrinsically parallel program where order has no relevance. Data dependencies between different tasks can severely limit parallel performance. If multiple cores need to write to the same place in memory at the same time some locking mechanism has to be applied, usually *mutexes* and/or *condition variables.* If all cores have the same number of tasks and each task takes approximately the same time, parallelization is trivial; otherwise, a more equitable work division has to be made and dynamic load balancing has to be taken into account.

In the subgraph counting problem, and most graph problems in general, work division is clearly not balanced. Graphs are completely irregular, with some nodes having a very high degree, such as *hubs*, and others having almost no connections. This means that, if a static work division strategy was adopted, a core with *computationally heavier* nodes would have much more work to do than other cores with *lighter* nodes. Therefore, a dynamic load balancing strategy is a requirement to achieve an efficient parallel subgraph counting algorithm.

3.1.1 Shared and Distributed Memory

The two main parallel paradigms are *distributed memory* and *shared memory.* In the distributed memory approach every core has a copy of the data in its own memory and works with it individually. On the other hand, with shared memory a unique copy of the data is used by all cores. Shared memory has the advantage of having less overhead, since data transfers are faster, but raises concerns on how memory is accessed. Distributed memory is more commonly adopted when multiple individual computers are used, such as in a cluster.

In this work we chose a shared memory approach, with our target being the multicore architecture. Multicore computers are now commonplace, with as many as 8 cores being widely available (and this number will surely increase in the coming years), giving us a broader scope than a distributed memory implementation that would require the users to have access to a dedicated cluster to be efficient. For this purpose we chose `Pthreads`, due to its flexibility and portability, being currently supported by all major operating systems.

3.1.2 Related Work

Although numerous sequential algorithms for network motif discovery do exist, parallel approaches are scarcer. Parallel distributed memory algorithms for both `ESU` [RSL12] and `gtrieScanner` [RSL10a] have been implemented, using MPI for communication. Our work here differs from those previous approaches because we instead aim for a shared memory environment with multiple cores. A different parallel algorithm is the one by Wang et al. [WTZ$^+$05], which employs a static pre-division of work and limits the analysis to a single network and a fixed number of cores (32). In our work, we instead apply dynamic load balancing and more thoroughly study the scalability of our approach. A subgraph-centric parallel algorithm using map-reduce was developed by Afrati et al. [AFU13], where they enumerate only one individual subgraph at a time. By contrast, we use two g-trie approaches, one set-centric and the other network-centric, and aim for a different target platform (multicores). For more specific types of subgraphs there are other parallel algorithms such as `Sahad` [ZWB$^+$12] (a hadoop subgraph-centric method for tree subgraphs), `Fascia` [SM13] (a multicore subgraph-centric method for approximate count of non-induced tree-like subgraphs) or `ParSE`. [ZKKM10] (approximate count for subgraphs that can be partitioned in two by a cut-edge), but our work stands apart by aiming at a more general set of subgraphs.

3.2 General Overview

Both algorithms start in the same way: each vertex in the input graph G is given as a candidate for the root node (lines 2 and 3 of Algorithm 2.1 for `gtrieScanner` and lines 3 and 4 of Algorithm 2.2 for `FaSE`). A naïve approach would be to simply divide these initial work units between the available computing resources. The problem with this static strategy is that the generated search tree is highly irregular and unbalanced, as discussed previously. A few of the vertices may take most of the computing time, leading to some resources being busy processing them for a long time while others are idle. To achieve a scalable approach for this kind of problem, we need an efficient dynamic sharing mechanism that redistributes work during execution time.

A major factor for both algorithms performance is that there is no explicit queue of unprocessed work units. Instead they are implicitly stored in the recursive stack. To achieve the best efficiency we kept this characteristic in our parallel approach as the queues would introduce a significant overhead both on the execution time and on the needed memory that would significantly deteriorate the sequential algorithms performance. Our goal is, therefore, to scale up our original efficient algorithm, providing the best possible overall running time.

We allocate one thread per core, with each thread being initially assigned an equal amount of vertices. When a thread P finishes its allotted computation, it requests new work from another active thread Q, which responds by first stopping its computation and then building a representation of its state, bottom-up, to enable sharing. Q proceeds by dividing the unprocessed work units in a round-robin fashion, achieving a *diagonal split* of the entire work tree, allowing itself to keep half of the work units and giving another half to P. Both threads then resume their execution, starting at the bottom (meaning the lowest levels of the g-trie) of their respective work trees. When all vertices for a certain g-trie node are computed, the thread moves up in the work tree. The execution starts at the bottom so that only one vector containing the current path is necessary, taking advantage of the common subtopology of ancestor and descendant nodes in the same path. When there is no more work, the threads terminate and the computed frequencies are aggregated. We will now describe in more detail the various components of our algorithms, starting by describing how each algorithm was adapted for a parallel execution and then detailing our general work sharing strategy.

3.3 Parallel Frequency Counting

3.3.1 Parallel `gtrieScanner`

Algorithm 3.1 depicts our parallel version of `gtrieScanner`. All threads start by executing `parallelCountAll()` with an initially empty work tree W (line 2). The first vertex that a thread computes is that of position $thread_{id}$ (lines 3 and 5). At each step, the thread computes the vertex $thread_{num}$ positions after the previous one (line 13). Every vertex is used as a candidate for the g-trie root node by some thread (lines 11 and 12). This division gives approximately $\frac{|V(G)|}{num\_threads}$ vertices for each thread to initially explore. We do this division in a round-robin fashion because it generally provides a more equitable initial division than simply allocating continuous intervals to each thread, due to the way we use the symmetry breaking conditions. Our intuition was verified empirically by observing that the threads would ask for work sooner if continuous intervals were used. When a thread Q receives a work request from P (line 6) it needs to stop its computation, saving what it still had left to do (line 7), divide the work tree (line 8), give P some work (line 9) and resume the remaining work (line 10). On the other hand, if a thread finishes its initially assigned work, it issues a work request to get new work (line 14).

Algorithm 3.1 The parallel `gtrieScanner` algorithm.

```
 1: procedure PARALLELCOUNTALL($T$, $G$)
 2:     $W \leftarrow \emptyset$
 3:     $i \leftarrow thread_{id}$
 4:     while $i \leq |V(G)|$ do
 5:         $v \leftarrow V(G)_i$
 6:         if WORKREQUEST($P$) then
 7:             $W$.ADDWORK()
 8:             $(W_Q, W_P) \leftarrow$ SPLITWORK(W)
 9:             GIVEWORK($W_P$, $P$)
10:             RESUMEWORK($W_Q$)
11:         for all children $c$ of $T.root$ do
12:             PARALLELCOUNT($c$, $\{v\}$)
13:         $i \leftarrow i + thread_{num}$
14:     ASKFORWORK()
15: procedure PARALLELCOUNT($T, V_{used}$)
16:     $V \leftarrow$ MATCHINGVERTICES($T, V_{used}$)
17:     for all vertex $v$ of $V$ do
18:         if WORKREQUEST($P$) then
19:             $W$.ADDWORK()
20:             return
21:         if $T.isLeaf$ then
22:             $thread_{freq}[T]$++
23:         else
24:             for all children $c$ of $T$ do
25:                 PARALLELCOUNT($c, V_{used} \cup \{v\}$)
```

`parallelCount()` remains almost the same as the sequential version, except for attending work requests and storing subgraph frequencies. If the thread receives a work request while computing matches, it first adds the vertices it still had to explore to the work tree W and then stops the current execution (lines 18 to 20) to compute the current state and finish building the work tree. In the sequential version we simply needed to increase the frequency of a certain subgraph in the g-trie structure. As for the parallel version, multiple threads may be computing frequencies for the same subgraph, using different vertices from the input graph, and so they need to coordinate the way they store the frequencies. Initially, we kept in each g-trie node a shared array $Fr[1..num\_threads]$ where the threads would update the array at the position of their $thread_{id}$. In the end, the global frequencies would be obtained by summing up the values in the array. This resulted in significant false sharing due to having too many threads update the frequency arays simultaneously, and became a severe bottleneck. Our solution was to create thread private arrays indexing g-trie nodes, i.e. $Fr[1..num_{gtrieNodes}]$, which impacted very favorably our efficiency. In our testing phase with a 24-core machine, we had cases with speedups below 5 that, just with this change, went to a speedup of over 22, thus converting a modest speedup into an almost linear one.

The `matchingVertices()` procedure remains the same as the sequential version, the only difference being that V_{used} is now thread local, with threads computing a different set of vertices.

3.3.2 Parallel `FaSE`

For `FaSE`, each V_s and V_{ext} pair can be regarded as composing a *work unit* and, along with the position in the g-trie, are sufficient to resume computation. At the start, V_s corresponds to each single node in the network and V_{ext} to its neighbors with higher index. We recall that `FaSE` creates the g-trie during execution, whereas `gtrieScanner` uses a previously built g-trie. We decided to use one central g-trie, as opposed to one g-trie per thread. While this option leads to contention when accessing the g-trie, it saves memory and removes the redundant work caused by multiple threads creating their own g-trie, with most connections being common for every thread. If a thread arrives at a new type of node it updates the g-trie. All threads see this change and do not need to update the g-trie if the node is found again.

Algorithm 3.2 details our parallel approach for `FaSE`. The graph G, the g-trie T and the subgraph size k are global variables, while *current* is a pointer to the g-trie location and is specific to each thread. Computation starts with an initially empty g-trie (line 2) and work queues (line 3), one for every thread. The condition in line 12 of Algorithm 2.2, $u > V_s[0]$, makes vertices with a smaller index more likely to be computationally heavier than higher indexed vertices. Because of this, network vertices are split in a round-robin fashion, giving all threads $V(G)|/num\_threads$ top vertices to initially explore (lines 4 to 6 and 13). This division is not necessarily balanced but finding the best possible division is as computationally heavy as the census itself. If a thread does not receive a work request it does the enumeration process starting at each of its assigned vertices (line 12). The `enumerate()` procedure is very similar to the sequential version but with V_s and V_{ext} now being thread local and the *count* variable becoming an array indexing threads, i.e. $count[thread_{id}]$, in each leaf. Another relevant difference is that, when a new node in the g-trie needs to be created, its parent node has to be locked before creation. This is done to ensure that the same node is not created by multiple threads. If a thread Q receives a work request from P, it needs to stop its computation, add the remaining work to W (line 8), split the work (line 9), give half the work to P (line 10) and resume its work (line 11). After the enumeration phase is finished, the leaves are also distributed among the threads and isomorphism tests are performed to verify the appropriate canonical type of each occurrence.

Algorithm 3.2 The parallel `FaSE` algorithm.

```
 1: procedure ParallelFaSE($G, T, k$)
 2:     $T \leftarrow \emptyset$
 3:     $W \leftarrow \emptyset$
 4:     $i, j \leftarrow thread_{id}$
 5:     while $i \leq |V(G)|$ do
 6:         $v \leftarrow V(G)_i$
 7:         if workRequest($P$) then
 8:             $W$.addWork()
 9:             $(W_Q, W_P) \leftarrow$ splitWork(W)
10:             giveWork($W_P$, $P$)
11:             resumeWork($W_Q$)
12:         enumerate($\{v\}, \{u \in N(v) : u > v\}, T.root$)
13:         $i \leftarrow i + num_{threads}$
14:     while $j \leq |T.leaves()|$ do
15:         $l \leftarrow T.leaves()_j$
16:         $frequency$[canonicalLabel($l.Graph$)] += $l.count$
17:         $j \leftarrow j + num_{threads}$
```

3.4 Work Sharing

We will now describe the work sharing process which can be divided in three main phases: *work request*, *work division* and *work resuming.* The process is very similar for both algorithms, only suffering minor changes due to the way the work units differ. A diagram with all thread states is shown in Figure 3.1 to depict the big picture of our approach.

3.4.1 Work Request

A work request is performed when some thread P has completed its assigned work. Since there is no efficient way of predicting exactly how much computation each active

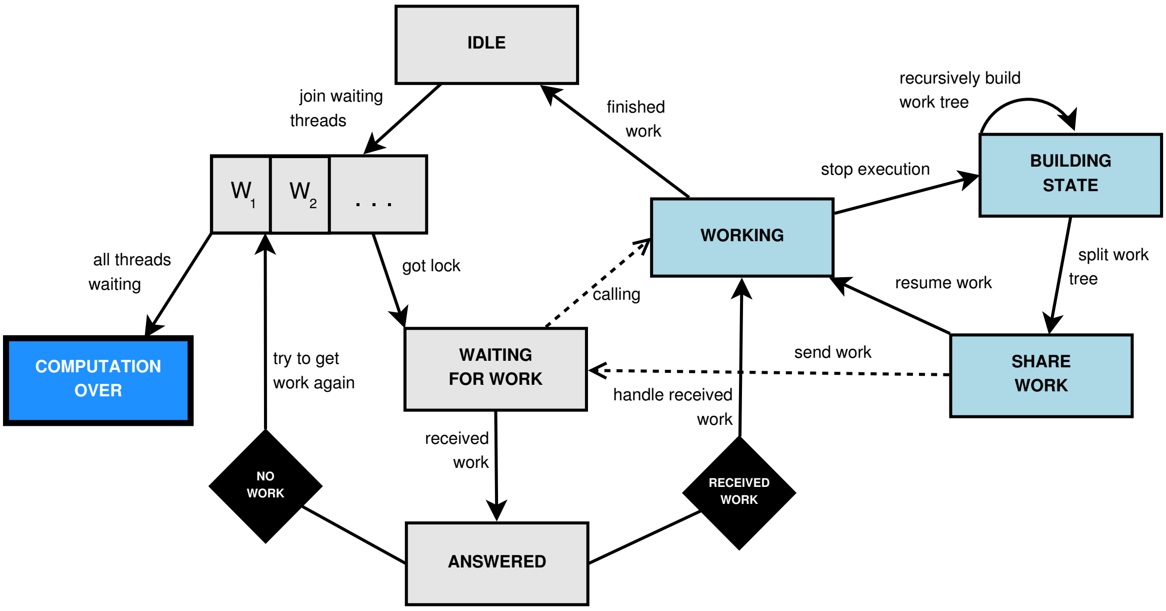


Figure 3.1: A complete state diagram of our parallel approach.

thread still has in its work tree, it asks a random thread Q for more work. Note that this kind of random polling has been established as an efficient heuristic for dynamic load balancing [San02]. If Q sends some unprocessed work, then P computes the work it was given. If Q did not have work to share, P tries asking another random thread. When all threads are trying to get work, no more work units are left to be computed and the enumeration phase ends. All this process of requesting work has to be protected by locks to ensure that a thread is only called by one other thread and that all requests are answered. To avoid busy waiting we use a conditional variable that is activated by thread Q to signal P that its request has been answered and it can proceed its execution with the new work queue.

3.4.2 Work Division

When a thread Q receives a work request it builds a work tree representing its current recursive state. In Figure 3.2 we show a resulting work tree and its division with a caller thread P, for `gtrieScanner`. The yellow colored circles constitute V_{used} and the yellow colored squares form the g-trie path up to the current level. The other nodes and vertices are still left to be explored and are split in a round-robin fashion. This division results in two work trees with approximately the same number of work units. This does not imply that two halves have the the same computational dimension, given the irregularity of the search tree they will induce, but nevertheless they constitute our best guess of a fair division across all levels.

As said before, we only build an explicit work tree when a work request is received. In that situation, a thread saves the current and the other unexplored vertices for the current node and moves up in the recursive tree. This process is repeated up to the top level, effectively populating the work tree with the unprocessed work units, i.e., the unexplored g-trie nodes and network vertices. This is a very fast operation and it is done by stopping the execution of the recursive `parallelCount()` calls and adding the work to the work tree (line 19 in Figure 2.1) until we get to `parallelCountAll()` and add the remaining nodes and vertices of the top level (line 7). We also store the current g-trie path and network vertices.

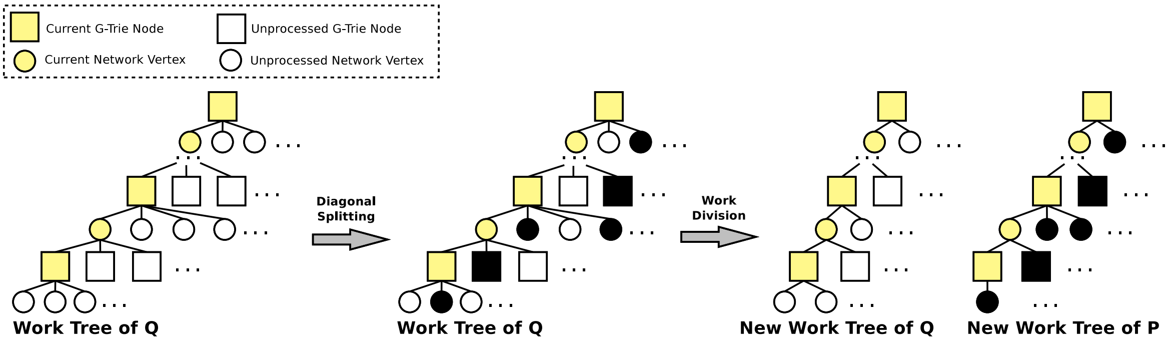


Figure 3.2: The constructed work tree and its division for `gtrieScanner` when a thread Q receives a work request from thread P.

The main difference in `FaSE` is that, during work division, each thread is given a complete g-trie level, constituted by V_s, V_{used} and the current g-trie position. For example, if a thread is stopped when it is in the fourth g-trie level, Q keeps level 3 and 1 while P receives 4 and 2. The topmost level is fully split since splitting it is the same as the initial division from lines 4 to 6 of Algorithm 3.2.

3.4.3 Work Resuming

After the threads have shared work, they need to resume their operation. We will describe how this process is applied to `gtrieScanner` and then point out the minor differences present in `FaSE`. First, the thread signals that it has not done any work yet (line 2), so that threads are not constantly sharing without advancing the computation. The work tree is then traversed in a bottom-up fashion (lines 3 to 6) and the vertices of each level are computed (line 7). If the thread receives a work request and has already done some work, work sharing is performed (line 8). There is no call to `addWork()` since the work is already on the work structure: either it was unfinished work already on W or it was added by the recursive `parallelCount()` calls. After having divided and shared the work (lines 9 and 10) the thread continues its computation with the new work tree (line 11) and the current execution is discarded (line 12). If the thread does not have pending work requests it proceeds to process the vertex. Thus, the thread has already done a minimum amount of work (line 13) and will attend work requests. Then the thread checks if it has arrived at a desired subgraph (line 14) and if so increases its frequency (line 15). If not, the thread calls `parallelCount()` with the new vertex in V_{used} for each child of the g-trie node (lines 17 and 18). When all work is completed it requests work from another thread (line 19).

The pseudo-code for `FaSE` is very similar, as shown in algorithm 3.4. The work levels are also ordered from top to bottom (line 2 and 3) so that only one V_s is necessary. If a work request is received, the general process of work sharing is performed (lines 4 to 8). No call to `addWork()` is necessary since the work was either added previously to W before the current `resumeWork()` call was made or was added by the recursive `addWork()` calls from `enumerate()`. If the level being computed is the root of the g-trie, the top vertices are individually computed (lines 9 to 11), in the same manner as line 12 of Algorithm 3.2. Otherwise, the stored values of V_s, V_{used} and *current* are used to continue the previously halted computation (lines 12 and 13). If the thread finishes its alloted work it asks for more (line 14).

Algorithm 3.3 Algorithm for resuming work after sharing is performed, applied to `gtrieScanner`.

```
 1: procedure RESUMEWORK($W$)
 2:     $did\_work \leftarrow false$
 3:     ORDERBYLOWEST($W$)
 4:     for all level $L$ of $W$ do
 5:         $depth \leftarrow L.depth - 1$
 6:         $V_{used} \leftarrow active\_vertices[1..depth]$
 7:         for all vertices $v$ of $L.nodes$ do
 8:             if WORKREQUEST($P$) and $did\_work$ then
 9:                 $(W_Q, W_P) \leftarrow$ SPLITWORK(W)
10:                 GIVEWORK($W_P$, $P$)
11:                 RESUMEWORK($W_Q$)
12:                 return
13:             $did\_work \leftarrow true$
14:             if $L.T.isLeaf$ then
15:                 $thread_{freq}[T]$++
16:             else
17:                 for all children $c$ of $L.T$ do
18:                     PARALLELCOUNT($c, V_{used} \cup \{v\}$)
19:     ASKFORWORK()
```

Algorithm 3.4 Algorithm for resuming work after sharing is performed, applied to FaSE.

```
procedure RESUMEWORK($W$)
    ORDERBYLOWEST($W$)
    for all level $L$ of $W$ do
        if WORKREQUEST($P$) then
            $(W_Q, W_P) \leftarrow$ SPLITWORK(W)
            GIVEWORK($W_P$, $P$)
            RESUMEWORK($W_Q$)
            return
        if $L.depth = 0$ then
            for all vertex $v$ of $L.V_{ext}$ do
                ENUMERATE($\{v\}, \{u \in N(v) : u > v\}, T.root$)
        else
            ENUMERATE($L.V_s, L.V_{ext}, L.current$)
    ASKFORWORK()
```

3.5 Obtaining the subgraph frequencies

In the end, both algorithms need to output the total number of occurrences of the queried subgraphs on the network and also the frequency of each subgraph type. The user may also choose to output the specific occurrences of each subgraph in the network.

In gtrieScanner the frequency of each subgraph type is stored by each thread in a private array indexing g-trie leaves (line 22 of Algorithm 3.1) and, in the end, the overall frequencies are aggregated by adding the values computed by every thread for each subgraph type.

As for FaSE, since we can not know how many leaves will eventually be created, the frequencies are actually stored in an array (indexing threads) in each g-trie leaf. The cycle from lines 14 to 17 of Algorithm 3.2 traverses all leaves to compute their canonical label to ensure that isomorphic graphs pertain to the same subgraph type. During the process the frequencies of every subgraph type computed by each thread are aggregated.

3.6 Summary

Here we discussed why we chose to parallelize g-trie based solutions and presented two parallel algorithms for subgraph counting based on two of the fastest sequential approaches for this same task. Both of them use a g-trie as their core but differ when the g-trie is created and have minor changes in what information they store. We also detailed our general dynamic work sharing strategy and applied it to both algorithms.

Parallel Subgraph Census using GPUs

4

The purpose of this chapter is to discuss the differences between the GPU and CPU architectures and why the GPU model makes it possible to improve the speed of existing algorithms. We present an algorithm for subgraph census based on g-tries that was implemented using `CUDA`. The chapter ends with a discussion of related work and the problems associated with graph traversal algorithms for this architecture.

4.1 Motivation

A *graphics processing unit* (GPU) is a device composed of a microchip specialized in visual output. They are highly efficient in processing mathematically-intensive tasks in parallel and can nowadays be found on video game consoles, personal computers and portable devices. The architecture of a GPU differs from that of a CPU by having hundreds of thousands of cores instead of one or just a few (Figure 4.1). This *many-core* architecture makes GPUs much faster than CPUs for tasks such as image processing that apply a single instruction to multiple data (SIMD).

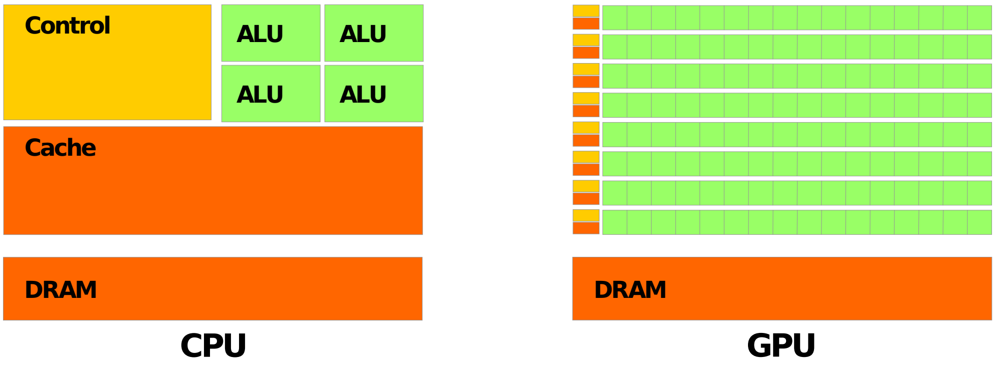


Figure 4.1: The CPU and GPU contrasting architectures (taken from `NVIDIA CUDA` C Programming Guide).

Despite their deep focus in graphics processing, the potential of GPUs offers programmers the opportunity to apply them to many different problems, such as sorting algorithms for very large lists or molecular dynamics simulations. This field was called *general-purpose computing on graphics processing units* (GPGPU), but efficiently mapping applications to the GPU proved to be too difficult. This task was made easier when `NVIDIA` developed the Tesla GPU architecture, that consisted of fully programmable processors with their own memory and a control logic similar to that of a CPU, and the `CUDA` framework to readily use the GPU in a relatively straightforward way. Currently, the dominant frameworks for GPU computing are `OpenCL` and `CUDA`. `CUDA` is unique to the `NVIDIA` graphics cards while `OpenCL` is open-source and supported by all major graphics card manufacturers, including `NVIDIA`. For `NVIDIA` cards, `CUDA` currently extracts better performances from the GPUs and we used it to implement our algorithm.

Modern `NVIDIA` GPUs consist of dozens of *streaming multiprocessors (SMP)* that are themselves split into dozens or hundreds of processors. Each SMP can manage thousands of hardware-scheduled threads. This physical structure is hidden from the typical programmer and is instead presented logically, as shown on Figure 4.2. A function to be executed on the GPU is called a *kernel* and is handled by a number of threads, called a *grid*. *Grids* are split into *blocks* that, themselves, contain the threads. Each thread has its unique set of *registers* and *local memory*, with an array of *shared memory* being common to threads of the same block. Threads of different blocks can not communicate between themselves and there is no synchronization of blocks, only threads of the same block can be synchronized. There are three other types of memory that all threads can access: *global memory*, which is a large memory that threads can read and write to, and two small read-only memories, *constant memory* and *texture memory*. Constant memory is particularly useful when there is data that never changes during the program's runtime and is accessed by all threads at the same position and at the same time. The CPU can transfer data in and out of these memories to make it available for all threads.

Rather than running individually, threads are grouped in *warps* of, usually, 32 threads that execute the same instruction at the same time. This limits the usage of branching code since having a branch forces each thread in the warp to execute both branches, adding non-productive computing time for each divergence in the code.

Using this architecture we developed an initial algorithm for subgraph counting based on g-tries that we will now describe.

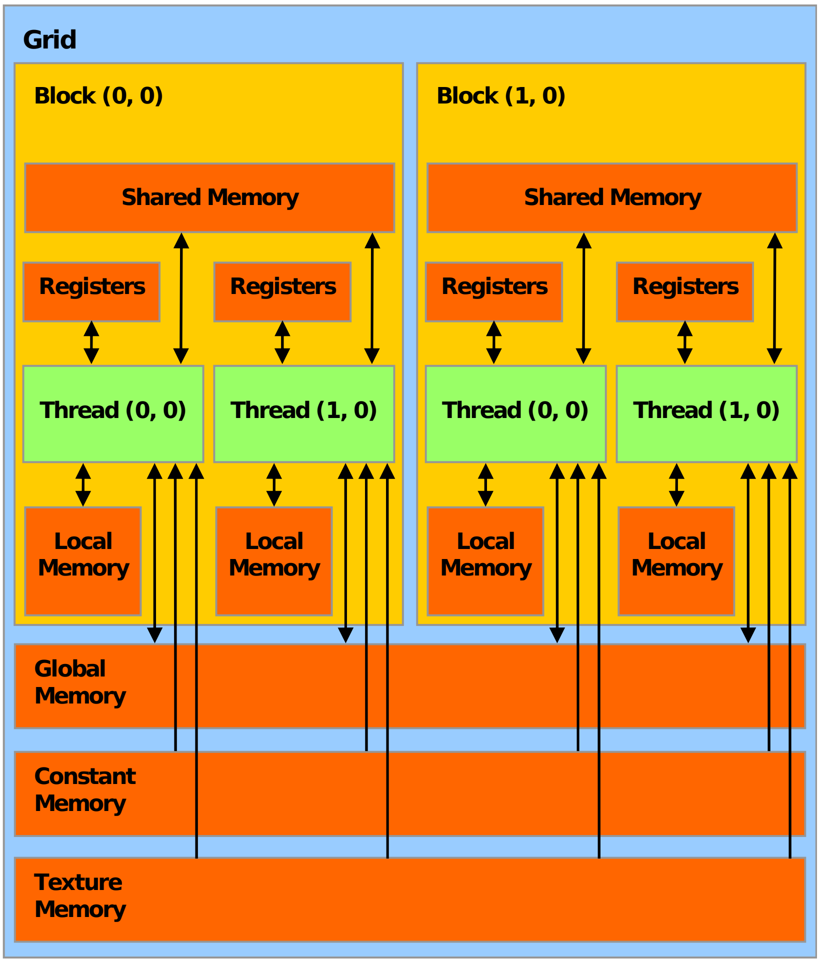


Figure 4.2: The CUDA programming model (taken from NVIDIA CUDA C Programming Guide).

4.2 GPU Algorithm

First of all, we need to decide what portion of the algorithm remains to be computed on the CPU and what portion is transferred to the GPU. In gtrieScanner, verifying if a found set of vertices corresponds to a valid subgraph, takes about 90% of the computing time. This corresponds to lines 17 and 18 of Algorithm 2.1, where the edge connections and symmetry conditions are evaluated. Thus, this is the section of the work that we want to send to the GPU to be executed in parallel.

Considering the enormous amount of matches the algorithm has to verify, sending one match at a time to the GPU is not practical due to the overhead of sending the work and receiving the results each time. Also, generally, the subgraphs that we want to search on the network are very small. GPU programming is ideal for executions with many threads, so we need a much bigger grain than a single graph verification.

In Figure 4.3 we have a partial search tree of a g-trie consisting of the size-3 undirected graphs in a small network. The lists of the form $[x_1, ..., x_k]$ are the currently matched nodes and the list $\{y_1, ..., y_n\}$ contains the candidates for expansion. A *work-unit* thus corresponds to a list $[x_1, ..., x_k]$ and one of the y_m associated to it. What the sequential version does is similar to a DFS, following the path: $(T_1, []) \rightarrow (T_2, [1]) \rightarrow (T_3, [1, 2]) \rightarrow (T_4, [1, 2]) \rightarrow (T_3, [1, 9]) \rightarrow (T_4, [1, 9]) \rightarrow (T_2, [2]) \rightarrow ...$, but it could do something closer to a BFS, completing a level of the g-trie before moving to the next. Adopting this method, we can send a large quantity of work-units to the GPU.

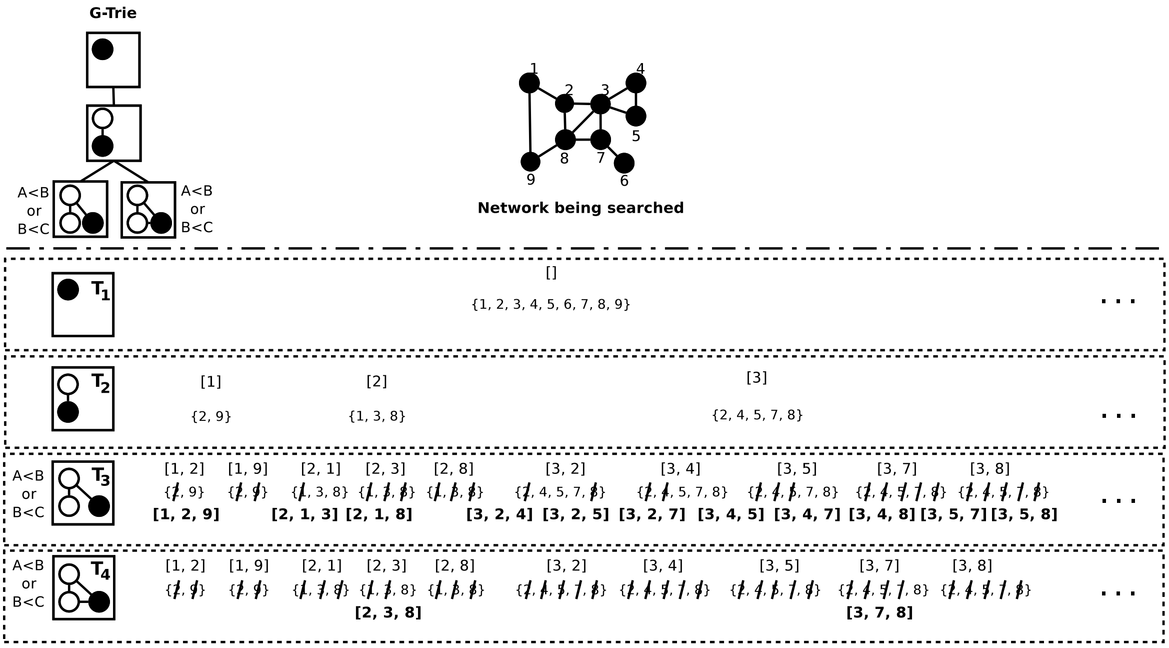


Figure 4.3: The `gtrieScanner` search tree.

Besides the g-trie and the input network, two lists, $MapList$ and $VertexList$, are sent to the GPU, the first containing all partial matches (or *maps*) for the current g-trie node and the latter the list of candidates, as illustrated in Figure 4.4. Each *map* points to its first associated vertex in $VertexList$. After the GPU threads are created, they receive k vertices from $VertexList$.

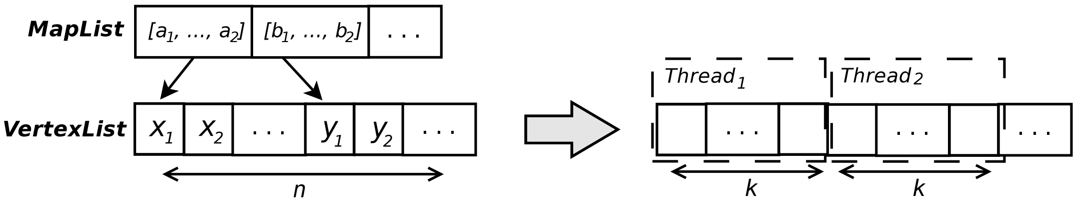


Figure 4.4: The work units assigned for each GPU thread.

Algorithm 4.1 GPU Algorithm for computing the frequency of subgraphs of g-trie T in graph G.

```
 1: procedure COUNTALL($T, G$)
 2:     $MapList \leftarrow V(G)$
 3:     $blocksize \leftarrow max\_threads\_per\_block$
 4:     $gridsize \leftarrow max\_concurrent\_threads/blocksize$
 5:     COUNT($MapList$, $T.root$)
 6: procedure COUNT($MapList, T$)
 7:     $size \leftarrow T.depth$
 8:     $VertexList \leftarrow \emptyset$
 9:     for $[x_i, ..., x_{size}]$ as $map$ in $MapList$ do
10:         $Vertices \leftarrow neighborVertices(T, map)$
11:         $Vertices.map \leftarrow map$
12:         $VertexList.add(Nodes)$
13:     DOMATCH<BLOCKSIZE, GRIDSIZE>($VertexList, MapList, T$)
14:     $newVertexList \leftarrow \emptyset$
15:     for all vertex $v$ of $VertexList$ that $matched$ do
16:         $newMapList.add([Map(v), v])$
17:         if $T.isLeaf$ then
18:             $T.frequency{+}{+}$
19:     for all child $c$ of $T$ do
20:         COUNT($newMapList, c$)
21: function NEIGHBORVERTICES($T, V_{used}$)
22:     $V_{conn} \leftarrow$ vertices connected to the vertex being added
23:     $m$     $\leftarrow$ vertex of $V_{conn}$ with smallest neighborhood
24:     $V_{cand} \leftarrow$ neighbors of $m$
25:     return $V_{cand}$
26: kernel DOMATCH<BLOCKSIZE, GRIDSIZE>($VertexList, MapList, T$)
27:     $k \leftarrow VertexList.size/total_{threads}$
28:     $i \leftarrow thread_{id} * k$
29:     while $i \leq (thread_{id} + 1) * k$
30:         $vertex \leftarrow VertexList[i]$
31:         $map \leftarrow Map(vertex)$
32:         $VertexList \leftarrow match([map, vertex], T)$
33:         $i{+}{+}$
```

Our GPU algorithm is depicted in Algorithm 4.1. The program begins by setting the initial $MapList$ as $V(G)$ (line 2). The block and grid sizes can be given as parameters but here we chose to use the `max_threads_per_block` as the blocksize and use the `max_concurrent_threads` as the total number of threads (lines 3 and 4). The `count()` procedure is then called to evaluate the $MapList$, starting at the g-trie root (line 5). In the `count()` procedure we start by checking the depth of the current g-trie node (line 7) and setting $VertexList$ as an empty list (line 8). Then, for each map from

MapList, we populate *VertexList* with the `neighborVertices(T, map)` (lines 9 to 12). Notice that `neighborVertices()` is the same function as `matchingVertices()` from `gtrieScanner` (Algorithm 2.1) but without asserting if the subgraphs actually correctly match the g-trie since that is the bulk of the work and it will be done in the GPU. The GPU kernel `doMatch()` is called with the given block and grid sizes, the work lists and the g-trie position (line 13). The work-units are equally split between the threads (lines 27 and 28) and each thread in the kernel traverses its portion of *VertexList* (lines 29 to 33). The threads verify if their respective alloted vertex (line 30) added to its partial match (line 31) - $[map, vertex]$ - respect the g-trie node connections and symmetry breaking conditions (line 32). The execution returns to the CPU that creates a list and populates it with the new partial matches (lines 15 and 16). If the match is actually a leaf, its frequency is incremented (lines 17 to 18). The whole process is repeated, matching *newMapList* to all descendant nodes of the current g-trie position (lines 19 and 20) until the g-trie is fully explored.

4.2.1 Memory Types

`CUDA` supports distinct kinds of memory that serve different purposes and using them correctly can boost the application's efficiency. Next we discuss where we use different kinds of memories and how they affect our performance.

Pinned Memory

In cases, such as ours, where data is transferred between CPU and GPU many times, the transactions between host and device can lead to a serious overhead and affect overall performance. In `CUDA`, when a regular memory transfer is issued, memory has to be transferred from pageable memory to pinned memory. This whole process involves creating a block of pinned memory, copying from pageable to pinned memory in the host, transferring the data from pinned memory to the GPU's RAM and deallocating the pinned memory. A better option is to allocate memory directly in pinned memory, allowing the GPU to transfer data from there without the intervention of the CPU. With this option, memory transfers become faster but allocation itself is slower. In our case we only need to allocate a large chunk of memory and use it when necessary. Even if we did not have to worry about the allocation overhead we would still need to limit the allocated memory since the number of work units can get so large that they do not fit memory. We solve this by not creating the exceeding vertices when the list limit is reached and proceeding with the computation to the next g-trie node. Eventually the program returns to the incomplete level and generates a list with the unexplored vertices.

Constant Memory

Constant memory is very useful when data never changes during the program's execution and all threads are accessing the same portion of memory because, if all threads from a warp read the same position from constant memory, only a single read is necessary. However, it is severely limited in size, with most GPUs having only a few dozen kilobytes. In our program, the g-trie is a good candidate for constant memory since it never changes during execution, all threads are computing the same g-trie node at the same time and is small enough to fit in this chunk of memory.

Global Memory

The global memory stores data that is common to all threads, such as the input network in the case of our algorithm. The network is generally too big to be put in constant memory and, as such, it is allocated in global memory.

Shared Memory

This is an additional type of memory that is shared between threads of the same block and has a faster bandwidth than global memory. The work lists $MapList$ and $VertexList$ are initially stored in global memory. If each *map* from $MapList$ has a number of vertices from $VertexList$ comparable to the number of threads per block, it would mean that threads in the same block likely share the same *map*. We explored this property by having a *leader thread* creating an array in shared memory, map_{block} composed of its *map*. Threads from the same block, that have the same *map*, can then use map_{block} instead of their own *map* from global memory. In practice, the effects are diminute because the subgraphs searched are small and, accordingly, so is the *map*.

4.3 Problems and Related Work

The GPU architecture achieves incredible speedups for problems with static and regular data, such as matrix-multiplication. However, adapting it for graph traversal problems is a challenging task due to their irregularity and data-dependency.

An approach by Pawan Harish and P.J. Narayanan [HN07] implemented a BFS solution using `CUDA` that gave a vertex to each thread. At each level of the BFS they had a *frontier array* with the vertices that were to be explored, with the source vertex being the only vertex in the *frontier* at the beginning. If the vertex of the thread was in the frontier, it would compute its vertex neighbors that were not yet explored. If it was not, the thread would do no work. This was an initial GPU approach that clearly did not achieve a balanced work division. They also tried a similar approach for *single shortest path* and *all pairs shortest path*. A more recent work, by Hong et al. in [HKOO11], tested various graph algorithms and tried to deal with work imbalance by using a warp-centric programming method that better uses the underlying GPU architectures and improves upon previous solutions. Still, the results achieved depend greatly on the benchmark they apply their solution to, with some cases with 15.1x speedup and others with virtually no speedup. In [MGG12], a BFS parallelization is offered and interesting speedups are achieved. However, the speedups still appear to depend on the network since the results range from 6x to 29x speedup. In the field of *frequent subgraph mining*, a GPU parallel implementation of `gSpan` [YH02] was made by Wang et al. [WDY13] which achieved speedups of about one order of magnitude when compared to the original `gSpan`. The results presented are for very small cases (less than a 100 vertices) and so it remains to be seen how well it would scale to bigger graphs.

4.4 Summary

We gave a general overview of the GPU programming model and its differences when compared to the traditional CPU architecture. An initial approach for subgraph counting using g-tries was presented and discussed. Finally, we exposed the reader to some related work and to the inherent difficulties to adapt graph traversal algorithms for the GPU.

Performance Evaluation 5

In this chapter we present empirical data obtained by running our parallel methods on a large and representative set of complex networks. Our purpose is to study the general scalability of the developed algorithms.

To study the multicore algorithms efficiency we first compare their original sequential version with our parallel implementations using only one thread. We then discuss the relative overhead of our sharing mechanism and end with scalability tests, showing the speedups we obtained.

We also present results for our proposed GPU algorithm in an effort to study how GPU architectures can fit the subgraph census problem. A comparison between our GPU approach and a modified sequential version is also put forward.

5.1 Common Materials

We gathered results for both multicore algorithms using the same computational environment. For the GPU tests we had to use a different machine with access to a high-end `CUDA`-capable GPU. The set of complex networks used for evaluation was also kept similar for all tests. We should note that, for brevity, not all networks are used in every test we performed.

5.1.1 Computational Environments

Our experimental results for both multicore algorithms were obtained on a 64-core machine, consisting of four 16-core AMD Opteron 6376 processors at 2.3GHz with a total of 252GB of memory installed. Each 16-core processor is split in two banks of eight cores, each with its own 6MB L3 cache. Each bank is then split into sets of two cores sharing a 2MB L2 and a 64KB L1 instruction cache. A 16KB L1 data cache is dedicated to each core. We disabled the turbo boost functionality because it would give us inconsistent results by having executions with less cores running at an

Device	Tesla C2050
`CUDA` **Cores**	448 (14 MP x 32 Cores)
Global Memory	2,687 MB
Constant Memory	64KB
Shared Memory	48KB
Warp Size	32 threads
Max. Threads p/MP	1,536
Max. Threads p/block	1024
Max. Concurrent Threads	21,504

Table 5.1: Our GPU's main characteristics.

increased clock rate. All code was developed in `C++11` and compiled using `gcc 4.8.2`. We used NPTL 2.18 for `Pthreads` support.

We performed the tests relative to our GPU approach on a 16-core machine consisting of four 4-core Intel Xeon E5620 processors at 2.4GHz with a total of 12GB of memory installed. The code was developed in `C++11`, using the latest `CUDA` driver version 6.0, and compiled with `nvcc 6.0.1`. The main characteristics of our GPU are shown in Table 5.1.

5.1.2 Networks

During our development phase we used a set consisting of a few dozen networks in an effort to guarantee that our parallel algorithms did not severely depend on the network structure. We will now describe the chosen representative subset of them and, in Table 5.2, give some general information concerning the dimension and type of the networks. In order to showcase the general scalability of our algorithm, we chose networks that vary in their field of application, their use of edge direction and their dimension, as can be seen in the aforementioned table.

- **Social Networks**: describe relations between users from social networks. These networks are becoming increasingly popular and studying their structure may give important insights into social organization [TMP12].
  - `facebook`: undirected network consisting of friend circles gathered from Facebook [ML12]. Source: [Les14].
  - `blogcat`: undirected network formed from friendship and group membership networks from BlogCat [TL09]. Source: [Uni14].

- **Collaboration Networks**: networks consisting of relations between entities collaborating in the same subject. Much attention has been given to co-authorship networks and in uncovering their underlying structure [Glä01, GS05].

  – `astroph`: undirected network of author collaborations on papers submitted to arXiv in the Astro Physics category [LKF07]. Source: [Les14].

  – `jazz`: undirected network composed of collaborations between jazz musicians from 1912 to 1940. [GD03]. Source: [Are14].

  – `netsc`: undirected network containing co-authorships of scientists working on network experiments and analysis [New06]. Source: [New10].

- **Communication Networks**: represent networks related to communications.

  – `polblogs`: directed network of hyperlinks between weblogs on United States politics [AG05]. Source: [New10].

  – `routes`: undirected network consisting of the traffic flow between routers [LKF05]. Source: [Les14].

  – `company`: directed network of ownership of media and telecommunication companies [NLGC02]. Source: [BM06].

  – `enron`: directed network aggregating around half a million emails [LLDM09]. Source: [Les14].

- **Biological Networks**: networks that model biological concepts. These networks are the most prevalent in the study of network motifs and their structure has be found to give important information, such as in the case of transcriptional regulation of *Escherichia coli* [SOMMA02]. They have also been important in the study of protein-protein-interaction [BZC$^+$03, CG08].

  – `ppi`: undirected network of protein-protein interaction between budding microorganisms (yeasts) [BZC$^+$03]. Source: [BM06].

  – `neural`: directed network of the nervous system of a small nematode (*C. elegans*) [WS98, WSTB86] Source: [New10].

  – `metabolic`: directed metabolic network of the same small nematode roundworm, *C. elegans* [DA05]. Source: [Are14].

Network	Group	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$\frac{\lvert E(G)\rvert}{\lvert V(G)\rvert}$	Directed
`jazz`	collaboration	198	2,742	13.85	No
`netsc`	collaboration	1,589	2,742	1.73	No
`ppi`	biological	2,361	6,646	2.81	No
`facebook`	social	4,039	88,234	21.85	No
`routes`	communication	6,474	12,572	1.94	No
`blogcat`	social	10,312	333,983	32.39	No
`astroph`	collaboration	18,772	198,050	10.55	No
`neural`	biological	297	2,345	7.90	Yes
`metabolic`	biological	453	2,025	4.47	Yes
`polblogs`	communication	1,491	19,022	12.76	Yes
`company`	communication	8,497	6,724	0.79	Yes
`enron`	communication	36,692	367,662	10.02	Yes

Table 5.2: The set of representative real networks used for parallel performance evaluation.

5.2 Multicore Algorithms

For the two multicore algorithms we compared the execution time of our version using only one thread with the original sequential algorithm to verify that our parallel solution does not impose a serious overhead. We also made extensive use of code profilers, such as `Intel VTune` and `AMD CodeXL`, to look for hotspots, particularly to study the overhead caused by our work sharing strategy. Finally, the speedups of our parallel algorithms are presented, to assess the scalability of our approach.

Having chosen the networks that will be queried, we also need to decide which subgraphs should be searched in those networks. For that purpose we use all possible subgraphs of a given size k, again to highlight general applicability. Note that when we consider directed networks, the number of possible subgraphs of size k increases drastically. For example, for $k = 4$ there are only 6 undirected graphs and 199 directed ones. One query on a directed network for $k = 4$ would thus imply counting the occurrences of 199 different types of subgraphs. Therefore, the chosen k for directed networks will, generally, be smaller than that of undirected networks in order to obtain more manageable execution times.

The g-trie sequential algorithms, `FaSE` and `gtrieScanner`, take a few seconds in cases where competing algorithms would take a considerable amount of time [RS14b, PR13]. Our purpose here is to explicitly pick very large cases even for g-tries. The sequential time for the examples used range from a couple of minutes to several hours. We chose this approach to show the real importance of our work, since going from a few seconds to tenths of seconds is of minimal practical interest to the user. Searching for larger subgraphs and using bigger networks takes longer but can provide new important insights and, from a practitioner point of view, our parallel approach increases the limits of what is feasible to compute in a reasonable amount of time.

5.2.1 `gtrieScanner`

Parallel Overhead

As said before, we wanted our parallel strategy with one thread to perform similarly to the original sequential version. Empirically we observed that our parallel implementation with one thread does not produce a high overhead, being less than 10% for all the networks we tested. The overhead lies mostly in threads having to check if they received a work request, with sharing itself having minimal impact. The results are shown on Table 5.3. Henceforth, we will use the *single thread time* as the *sequential time* and use it to measure speedups. This means that our speedups are *relative speedups* and not *absolute speedups*. Nevertheless, the overhead is sufficiently low to give a clear idea of the actual gain.

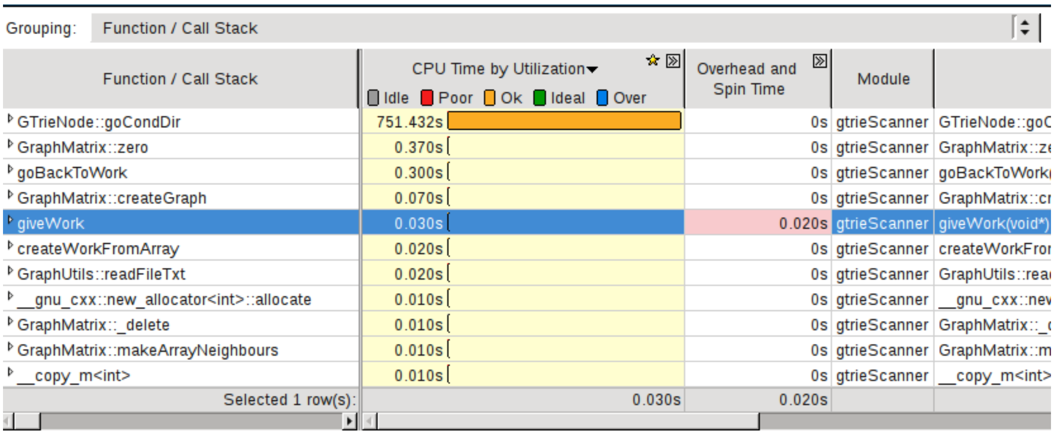
Network	Directed-Size	Sequential Time (s)	Single Thread Time (s)	Overhead
`netsc`	undir-9	463.77	466.48	≈ 1%
`facebook`	undir-5	6,001.79	6,043.90	≈ 7%
`routes`	undir-5	4,824.76	4,936.54	≈ 2%
`blogcat`	undir-4	5,204.64	5,410.45	≈ 4%
`metabolic`	dir-6	532.03	580.28	≈ 9%
`polblogs`	dir-5	985.23	1,018.27	≈ 3%
`company`	dir-5	212.89	220.45	≈ 4%
`enron`	dir-4	973.82	1,038.60	≈ 7%

Table 5.3: `gtrieScanner`: Comparison between the original sequential version and the parallel version with one thread.

Work Sharing

Using code profilers, such as `Intel VTune` and `AMD CodeXL`, we verified that sharing took a negligible amount of time (less than 1% of the total time), as can be verified in Figure 5.1. This gives us strong evidence that our dynamic workload balancing mechanism is extremely lightweight when compared to the actual subgraph counting process itself, substantiating its effectiveness. Thus, our mechanism is able to quickly divide and share the work between threads and the diagonal task splitting gives a probably balanced division that reduces the amount of times that work needs to be shared.

In Figure 5.2 the communication between threads is represented by yellow lines connecting two threads and the requester thread is identified by a yellow dot. As the figure shows, more threads communicate nearing the end of the computation since the work trees become smaller and, accordingly, the threads finish their work faster, resulting in work requests being sent in increasingly smaller time intervals.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization ▢ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Overhead and Spin Time	Module	
GTrieNode::goCondDir	751.432s	0s	gtrieScanner	GTrieNode::goC
GraphMatrix::zero	0.370s	0s	gtrieScanner	GraphMatrix::ze
goBackToWork	0.300s	0s	gtrieScanner	goBackToWork
GraphMatrix::createGraph	0.070s	0s	gtrieScanner	GraphMatrix::cr
giveWork	0.030s	0.020s	gtrieScanner	giveWork(void*)
createWorkFromArray	0.020s	0s	gtrieScanner	createWorkFror
GraphUtils::readFileTxt	0.020s	0s	gtrieScanner	GraphUtils::rea
__gnu_cxx::new_allocator<int>::allocate	0.010s	0s	gtrieScanner	__gnu_cxx::nev
GraphMatrix::_delete	0.010s	0s	gtrieScanner	GraphMatrix::_
GraphMatrix::makeArrayNeighbours	0.010s	0s	gtrieScanner	GraphMatrix::m
__copy_m<int>	0.010s	0s	gtrieScanner	__copy_m<int>
Selected 1 row(s):	0.030s	0.020s		

Figure 5.1: A screen capture from `Intel VTune` showing relative sharing time.

Speedup

Our algorithm was evaluated up to 64 cores. As mentioned before, we searched in the network for all possible graphs of a given size k. In Table 5.4 we show the size k used and the resulting number of all possible subgraphs of that size and type (directed or undirected) that will be counted in that network. The sequential time and the obtained speedups for 8, 16, 32 and 64 cores are shown in Tables 5.5 and 5.6. We present two tables containing the speedups with and without compiler optimization (gcc `-O0` and `-O3` flags, respectively) because we observed significant differences in the results. This happens due to some compiler optimizations that are effective for sequential programs not being designed for parallel programs. For example, some cache optimizations that greatly reduce the sequential time do not work as well when multiple cores are running at the same time. This effect may cause an unfair comparison between sequential and parallel executions. Nevertheless, results from Table 5.5 are also positive and users will be more interested in real execution times than speedups, therefore we decided to include both tables for the sake of completeness.

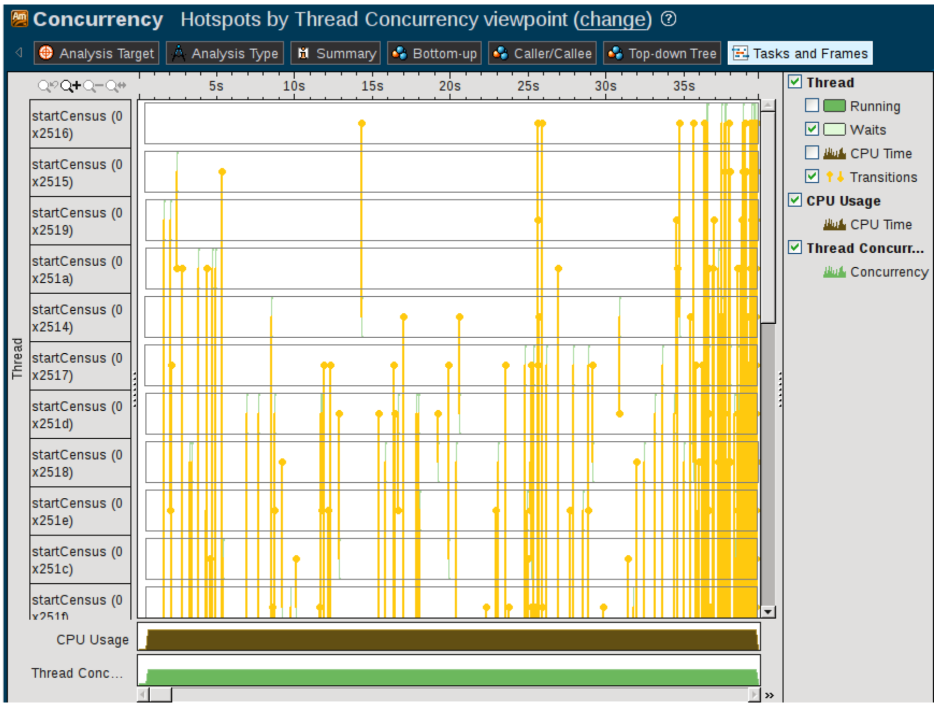


Figure 5.2: A screen capture from `Intel VTune` showing thread communication.

The results we obtained are very promising and up to 32 cores we achieved near-linear speedup, for both directed and undirected networks. With 64 cores we still achieve over 75% efficiency. We should reassert that each pair of cores shares its 2MB L2 and 64KB L1 instruction cache. This makes it harder to obtain perfect linear speedup because these cores are not completely independent. For testing purposes, we experimented with the well known `pbzip`[5] parallel data compression algorithm, which should achieve near-linear speedup on shared memory machines. Nevertheless, `pbzip` had a performance similar to our own algorithm, with near-linear speedup up to 32 cores and with a speedup of around 50 for 64 cores, further substantiating the idea that, with a different architecture, our algorithm could still present near-linear speedup with more than 32 cores.

We can also observe that as the network size increases, the performance slightly degrades. This is particularly noticeable in the two largest networks, which show the worst behavior. This is mostly due to their large size leading to memory constraints and cache issues. Note, however, that their behavior without compiler optimizations is not significantly worse. Furthermore, we used an adjacency matrix to represent the network. This gives the best possible algorithmic complexity for verifying if an edge exists but, at the same time, imposes a quadratic representation in memory. Other

[5] Parallel BZIP2 (PBZIP2): http://compression.ca/pbzip2/

Network	Subgraph size	#Subgraphs searched
`netsc`	9	261,080
`facebook`	5	21
`routes`	5	21
`blogcat`	4	6
`metabolic`	6	1,530,843
`polblogs`	6	1,530,843
`company`	6	1,530,843
`enron`	4	199

Table 5.4: Overall execution information for `gtrieScanner`.

Network	Sequential time (s)	#Threads: speedup			
		8	16	32	64
`netsc`	466.48	**7.90**	**15.78**	**30.91**	**51.09**
`facebook`	6,043.90	**6.75**	**14.72**	**30.23**	**52.47**
`routes`	4,936.54	**6.53**	**14.52**	**30.34**	**48.76**
`blogcat`	5,410.45	**7.72**	**14.37**	**24.92**	**25.69**
`metabolic`	580.28	**6.38**	**14.12**	**29.46**	**40.44**
`polblogs`	91,190.73	**7.87**	**15.69**	**31.31**	**52.96**
`company`	26,955.71	**6.74**	**14.54**	**29.99**	**45.12**
`enron`	1,038.60	**6.23**	**12.69**	**23.78**	**24.41**

Table 5.5: Results with compiler optimizations for `gtrieScanner`.

Network	Sequential time (s)	#Threads: speedup			
		8	16	32	64
`netsc`	2,030.39	**7.91**	**15.74**	**31.36**	**51.65**
`facebook`	17,851.16	**6.78**	**14.67**	**30.31**	**53.84**
`routes`	20,706.67	**6.80**	**14.67**	**30.53**	**52.44**
`blogcat`	15,666.05	**7.88**	**15.40**	**29.60**	**48.69**
`metabolic`	1,920.41	**6.61**	**14.44**	**30.18**	**49.73**
`polblogs`	222,210.76	**7.91**	**15.78**	**31.38**	**52.11**
`company`	94,384.39	**6.69**	**14.61**	**30.17**	**47.09**
`enron`	2,768.74	**6.42**	**13.69**	**27.43**	**45.59**

Table 5.6: Results without compiler optimizations for `gtrieScanner`.

data structures would degrade edge verification performance but also significantly decrease the memory footprint, and thus would contribute to a potentially more scalable shared memory parallel performance. We should also note that previous work has been done to parallelize `gtrieScanner` in a distributed memory environment which obtained near-linear speedup up to 128 processors [RSL10a]. In that architecture, each CPU has its own dedicated main memory storing a copy of the graph, which means that the problems related with competing memory and caching are not present. However, the typical practitioner may not have access to a dedicated cluster, limiting that approach's scope. Using our implementation, any user can run the algorithm and have it run 2, 4 or 8 times faster, depending on the user's machine number of cores.

5.2.2 FaSE

Following the same idea that we used to evaluate `gtrieScanner`, we decided to find all graphs of a given size k that gave a sufficiently large sequential time for parallelism to be meaningful but not so large that it would take more than a few hours to complete the computation. `FaSE`, in general, is slower than `gtrieScanner` and, for some cases, we reduced k in order to have a more manageable reproducibility.

Parallel Overhead

As with `gtrieScanner`, in order to have our parallel version with one thread perform similarly to the sequential algorithm we did not artificially create work queues. This choice led us to have a very small overhead (no more than 6% for all tested cases) and, from now onwards, we will use the parallel execution with one thread as the *sequential time.*

Network	Directed-Size	Sequential Time (s)	Single Thread Time (s)	Overhead
`jazz`	undir-6	291.68	295.95	≈ 1%
`netsc`	undir-9	288.17	295.12	≈ 2%
`facebook`	undir-5	3,402.01	3,598.41	≈ 6%
`astroph`	undir-4	169.49	179.47	≈ 6%
`polblogs`	dir-5	1,734.64	1,722.55	≈ 0%
`company`	dir-5	739.07	739.12	≈ 0%
`enron`	dir-4	1,287.64	1,370.46	≈ 6%

Table 5.7: `FaSE`: Comparison between the original sequential version and the parallel version with one thread.

Locking and Work Sharing

In FaSE the g-trie is created on-the-fly, with new nodes (corresponding to subgraphs) being inserted only when they appear for the first time in the network. Since the algorithm has multiple threads updating the g-trie, a synchronization mechanism needs to be enforced in order to create a consistent g-trie. In a first approach we used a lock to fully protect the g-trie: every time a thread wanted to insert a new node, a global lock was made keeping other threads from adding new nodes. This strategy caused a severe overhead, most significantly when k was large (8 or above) and for directed networks because the total number of created nodes is in the order of the millions or billions, increasing the time threads spent waiting for the lock. The best possible option would be to use one lock per node, that way a thread would only get locked if another thread was creating a child node precisely in the same g-trie node. Unfortunately, we do not know the total number of nodes that will effectively be created and using the total number of possible nodes is impractical. Another option is to use *lock striping*, composed of an array of 10^5 or 10^6 indices, and mapping each g-trie node in a position of the array of locks. This choice, however, has different nodes mapping to the same array position, therefore making the lock unnecessary but, in practice, works fairly well if the key for each node is random enough. For that purpose we simply use the pointer of the node. For example, if we have an array of 500,000 locks and the pointer to the current node is $0 \times b8000000$, it would map to position $\lceil dec(0 \times b8000000)/500000 \rceil = 6174$. We show the impact of this change in Figure 5.3 where the red colored areas represent locking intervals and with this change we can clearly see that the red areas greatly diminished.

Each time a thread arrives at a leaf, it needs to verify its canonical representation. If the leaf does not correspond to an already matched class, a new label has to be created so that we can start counting subgraph occurrences for that class. For isomorphism tests we use the external nauty module to which we had to make minor changes in order to support multi-threading. If indeed a new node has to be created, a global lock is made on the g-trie. This lock does not interfere with the locks to insert new nodes, since inserting a new node and creating a label are totally independent tasks. However, this means that only one thread can identify a new class at a time. We can not adopt a similar strategy to the one we used for node insertion since we need to keep a global leaf counter, and having multiple threads updating it would lead to inconsistency. However, finding new classes is a relatively rare occurrence when doing subgraph census. For example, for one of our test cases that takes in total over 300 seconds, only 4 seconds are spent on the lock, as shown on Figure 5.4.

Work sharing itself, as we observed for `gtrieScanner` takes a negligible amount of time (less than 1%), again highlighting the effectiveness of our workload balancing mechanism and demonstrating that it can be successfully applied to different algorithms.

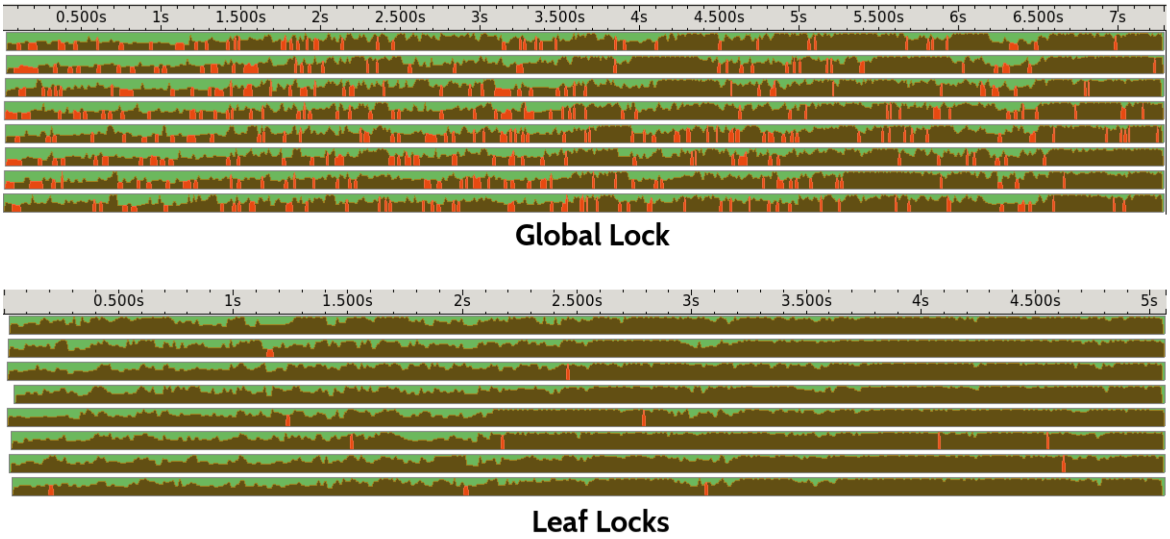
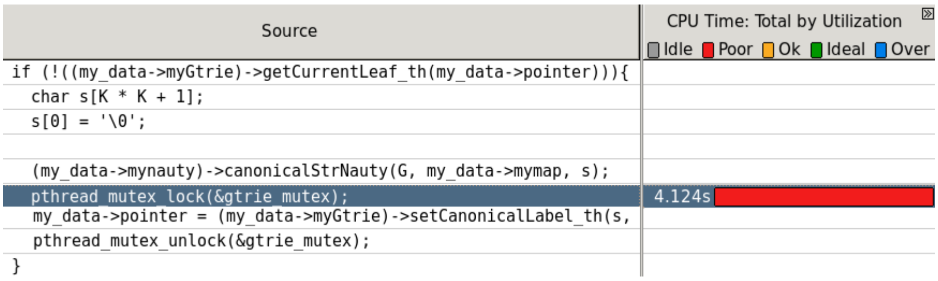


Figure 5.3: A screen capture from `Intel VTune` comparing the impact of a global lock versus lock striping when inserting a new g-trie node.


Source
CPU Time: Total by Utilization
▢ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over
4.124s

```
if (!((my_data->myGtrie)->getCurrentLeaf_th(my_data->pointer))){
  char s[K * K + 1];
  s[0] = '\0';

  (my_data->mynauty)->canonicalStrNauty(G, my_data->mymap, s);
  pthread_mutex_lock(&gtrie_mutex);
  my_data->pointer = (my_data->myGtrie)->setCanonicalLabel_th(s,
  pthread_mutex_unlock(&gtrie_mutex);
}
```

Figure 5.4: A screen capture from `Intel VTune` showing time spent on a lock for label insertion in the g-trie.

Speedup

We ran our algorithm up to 64 cores although obtaining near-linear for that number of cores is not possible due to the machine's cache architecture, as explained before. The turbo boost functionality was again disabled and we show the results with compiler optimizations in Table 5.9 and without them in Table 5.10.

We show the size of the subgraphs being queried, along with the number of g-trie leaves (the intermediate classes) and the actual number of different subgraph types in

Network	**Subgraph size**	**#Leafs found**	**#Subgraphs types found**
jazz	6	3,113	112
netsc	9	445,410	14,151
routes	5	125	19
blogcat	4	17	6
polblogs	5	409,845	9,360
company	5	1,379	310
enron	4	17	6

Table 5.8: Overall execution information for FaSE.

Network	**Sequential time (s)**	**#Threads: speedup**			
		8	16	32	64
jazz	121.19	**6.20**	**14.34**	**28.71**	**44.89**
netsc	122.02	**7.66**	**13.76**	**16.83**	**15.54**
facebook	1,358.97	**7.60**	**15.50**	**31.01**	**46.43**
astroph	93.46	**6.15**	**12.49**	**20.77**	**19.35**
polblog	801.94	**7.81**	**15.23**	**28.34**	**37.74**
company	319.71	**7.91**	**12.59**	**30.57**	**43.50**
enron	710.02	**7.46**	**12.39**	**22.99**	**24.33**

Table 5.9: Results with compiler optimizations for FaSE.

Network	**Sequential time (s)**	**#Threads: speedup**			
		8	16	32	64
jazz	295.95	**6.75**	**14.86**	**29.92**	**49.74**
netsc	295.12	**7.83**	**15.05**	**23.82**	**26.54**
facebook	3,598.41	**7.67**	**15.34**	**31.00**	**51.81**
astroph	179.47	**6.62**	**13.60**	**24.69**	**30.42**
polblogs	1,722.55	**7.85**	**15.56**	**30.04**	**47.48**
company	739.12	**7.94**	**15.81**	**31.02**	**48.53**
enron	1,370.46	**7.70**	**13.32**	**25.44**	**35.85**

Table 5.10: Results without compiler optimizations for FaSE.

Table 5.8. The sequential time and the obtained speedups for 8, 16, 32 and 64 cores are also shown.

The results we obtained are promising and achieved almost linear speedup for most cases. We should observe that our algorithm performs worse in networks where many leaves need to be created. This problem arises because we use a unique g-trie and need to protect it when a new node (or leaf) is inserted and when a thread tries to insert a new label, as elaborated on previously. We found cases where our speedups were severely limited by this fact. On the other hand, using one g-trie per thread would lead to a lot of redundant work that would deteriorate our algorithm's performance. Memory also becomes a concern when many threads are used because each leaf has an array to keep the frequencies. This limits the size of the subgraphs and networks that can be run. Another problem comes from the way we store the frequencies in the g-trie since it can sometimes lead to false sharing when too many threads are trying to update the array at the same time. A better option would be to, instead, have each thread keep an array of the frequencies for each leaf but, since the g-trie is created during execution, we can not know the total number of leaves and set a unique *id* in each one without resorting to locks. Finally, we observed that memory allocations became heavier when more threads are used. Something we intend to explore is an efficient pre-allocation of memory, where the threads would retrieve memory when needed. Similarly to `gtrieScanner`, we used an adjacency matrix to represent the input network that, while giving the best possible algorithmic complexity for verifying node connections, imposes a quadratic memory representation. We also tried different memory allocators, such as `jemalloc` and `tcmalloc`, but found no significant and consistent performance improvement.

By comparison, `gtrieScanner` obtained almost linear speedup for every case we tested. Besides using a conceptually different base approach (here we follow a network-centric algorithm), the main difference between the two algorithms is that, for `gtrieScanner`, the g-trie is created before subgraph counting, removing the need to have locks when modifying the g-trie and making it possible to have subgraph frequencies outside of the g-trie, eliminating false sharing.

5.3 GPU Approach

We did some initial experimentation using `CUDA` for subgraph counting and here present the results obtained for the algorithm discussed in Chapter 4.

5.3.1 Thread and Work List Sizes

To have an amount of work units comparable to the number of threads GPUs can offer, we chose to transfer all possible matches of a given g-trie node to the GPU. For big networks this number can be so large that it does not fit in memory, so we had to chose a proper size for the maximum number of work units in the work list.

At the same time, we want our program to have an efficient *occupancy.* Occupancy is the relation between active and maximum active warps, for which having a value between 60 and 70% is recommended. There are factors that limit occupancy such as registers (only 32 thousand per streaming multiprocessor) or shared memory (only a total of 16 to 48 KB depending on the GPU). The number of threads is also a limiting factor, with the number of *concurrent threads* being restrained by the number of streaming multiprocessors (SMP), the number of resident warps and the number of threads each warp is constituted by, as shown in Equation 5.1. Our Tesla GPU has 14 SMPs, 42 resident warps and 32 threads per warp, giving a total of 21,504 maximum concurrent threads.

$$max_{conc\_threads} = N_{SMP} * N_{Warps} * Warp_{size} \tag{5.1}$$

For all results presented here we set the *block size* as the maximum number of threads that out GPU supported (512). This number should always be a multiple of 32, due to the way the threads are managed in warps, and big enough to achieve a good occupancy, since the number of active blocks per streaming processor is limited.

To study the best size for the work list and the optimal number of threads we used multiples of $max_{conc\_threads}$. So, if we have ∼21 thousand threads and ∼86 thousand work units, each thread will get 4 units to compute. We present the results in Table 5.11 with the values inside parentheses being the number of work units per thread. Looking at the table we see that the total number of threads does not make much difference in the execution times, but rather the number of work units per thread does.

Giving one work unit per thread yields the best results, regardless of the total number of threads. This may happen since using more than one thread leads to branching problems inside the same warp, with threads still executing some work unit while others already moved to the next one because the first unit failed some matching test (symmetry conditions or ancestor connections). For each branch in a warp, the work is therefore doubled because all threads need to execute both sides of the branch and ignore the results of the incorrect path.

Using more than the maximum number of concurrent threads gives slightly faster times for some cases. This may be because the time spent creating threads is so small that, in practice, it becomes irrelevant that only some threads can execute at the same time. On the other hand using, for instance, ∼21 thousand instead of ∼86 thousand threads has 3 times the overhead of creating the kernel and waiting for it to execute, which degrades the executing time.

The results also show that using up to 16 work units per thread gives better results, but after 32 work units per thread the performance becomes worse. Again, this may happen because of the overhead caused by entering and waiting for the kernel to execute. Executions with fewer threads have a relatively larger kernel execution overhead, while executions with many threads have more problems with branching code. A better balance is achieved by using 16 units per thread. Since using one thread has poses no branching problems, its performance is the best despite the kernel execution overhead.

Threads **Units**	21k	43k	86k	172k	344k
21k	13	–	–	–	–
43k	95(*2*)	12	–	–	–
86k	83(*4*)	96(*2*)	12	–	–
172k	62(*8*)	84(*4*)	96(*2*)	13	–
344k	24(*16*)	62(*8*)	84(*4*)	96(*2*)	13
688k	32(*32*)	25(*16*)	63(*8*)	85(*4*)	96(*2*)

Threads **Units**	21k	43k	86k	172k	344k
21k	364	–	–	–	–
43k	1,335(*2*)	358	–	–	–
86k	1,144(*4*)	1,327(*2*)	356	–	–
172k	826(*8*)	1,142(*4*)	1,326(*2*)	356	–
344k	360(*16*)	828(*8*)	1,144(*4*)	1,328(*2*)	362
688k	489(*32*)	359(*16*)	833(*8*)	1,149(*4*)	1,134(*2*)

Table 5.11: Execution times of our GPU algorithm using different list sizes (units) and number of threads for 2 networks (`blogcat` and `neural`).

5.3.2 Comparison with CPU version

Since our GPU algorithm creates a work list for every g-trie node, whereas the original CPU sequential algorithm for `gtrieScanner` did not, the overhead introduced limits our algorithm's performance. We therefore implemented a sequential algorithm that performs the same tasks as the GPU alternative: (i) creation of the work lists, (ii) doing the matching and (iii) checking if the matches were valid. We did this to get a better feel of the actual gains we obtain by running the matching process in the GPU. The comparative results are displayed in Table 5.12.

The comparison between our modified CPU sequential algorithm and the GPU version is shown in Table 5.13, with the same subgraph sizes being used. We ran ∼86 thousand units for the list size as well as the total number of threads, because they were the parameters that achieved the best results.

Network	Subgraph size	Original Version Time (s)	Adapted Version Time (s)	Overhead
jazz	6	116.23	273.64	2.35x
facebook	5	370.78	2,111.43	5.69x
routes	5	2,970.42	14,602.85	4.92x
blogcat	4	2,705.38	8,209.92	3.03x
astroph	4	29.79	79.19	2.66x

Table 5.12: Comparison between the original `gtrieScanner` and an adapted sequential version.

Network	Work Creation (s)		Matching (s)		Checking (s)		Total Time (s)	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
jazz	10.73	11.91	205.72	1,371.55	49.63	52.90	273.64	1,449.25
facebook	55.34	79.99	1,388.11	386.22	670.11	605.28	2,111.43	1,130.34
routes	311.06	622.36	8,572.37	2,839.55	4,245.15	4,984.36	14,062.85	8,930.80
blogcat	185.77	256.80	5,577.52	7,330.96	2,550.83	2,628.23	8,209.92	10,377.47
astroph	2.40	2.77	52.20	134.57	23.33	22.19	79.19	164.19

Table 5.13: Comparison between CPU and GPU execution times.

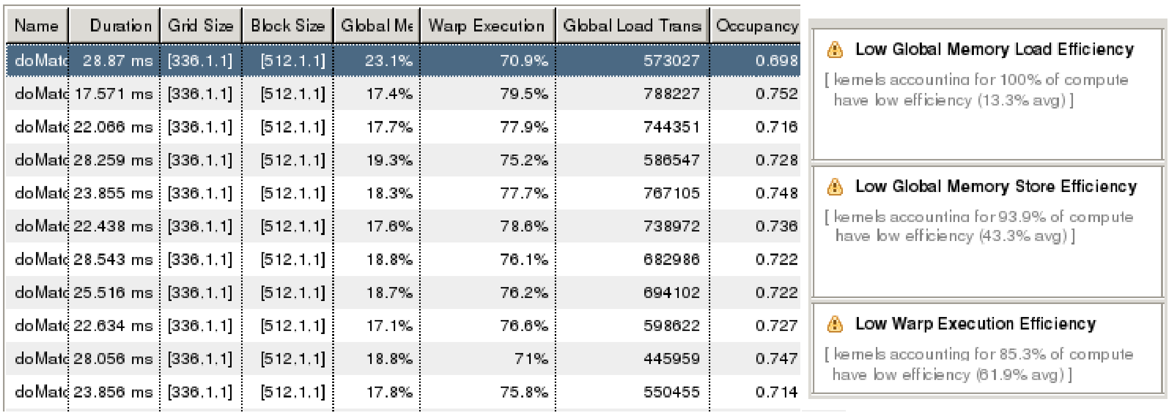
We separated the execution times of the three aforementioned tasks. For the *work creation* and *checking* tasks it is expected to have an overhead on the GPU alternative since the memory has to be made available to the GPU. As for the core of the algorithm, the *matching* process, the results depend on the graph. There are cases, such as `facebook` and `routes`, where the GPU algorithm is faster but others where it is slower.

To understand why this is, we used the profiler `nvpp` that comes with the `CUDA SDK`. We show an adapted screen-shot, for space reasons, from `nvpp` in Figure 5.5. It informs us that our implementation has two major problems: memory transfers and warp execution lack of efficiency.

The first problem is caused by the accesses to global memory which are not *coalesced*, with nearby threads possibly checking vertices from totally distinct positions in memory. This is an inherent problems in graph traversing since we can not separate the graph in sections and split them among threads. We could try to do some preprocessing on the graph, in an effort to get closely numbered labels to nearby vertices but the efficiency of that strategy would depend on the graph.

Warp execution efficiency is lowered when branching code is used. For example, if a thread fails the symmetry conditions it does not need to process the connections to ancestors, and therefore it creates a new branch. In a similar way if a thread fails one of the connections it does not need to process any further connections.

The occupancy is high since we are using a block size that does not limit the number of active blocks, a low amount of registers (15 out of a possible 63, for our GPU) and we do not surpass the amount of shared memory per block.



Name	Duration	Grid Size	Block Size	Global Me	Warp Execution	Global Load Trans	Occupancy
doMatc	28.87 ms	[336.1.1]	[512.1.1]	23.1%	70.9%	573027	0.698
doMatc	17.571 ms	[336.1.1]	[512.1.1]	17.4%	79.5%	788227	0.752
doMatc	22.066 ms	[336.1.1]	[512.1.1]	17.7%	77.9%	744351	0.716
doMatc	28.259 ms	[336.1.1]	[512.1.1]	19.3%	75.2%	586547	0.728
doMatc	23.855 ms	[336.1.1]	[512.1.1]	18.3%	77.7%	767105	0.748
doMatc	22.438 ms	[336.1.1]	[512.1.1]	17.6%	78.6%	738972	0.736
doMatc	28.543 ms	[336.1.1]	[512.1.1]	18.8%	76.1%	682986	0.722
doMatc	25.516 ms	[336.1.1]	[512.1.1]	18.7%	76.2%	694102	0.722
doMatc	22.634 ms	[336.1.1]	[512.1.1]	17.1%	76.6%	598622	0.727
doMatc	28.056 ms	[336.1.1]	[512.1.1]	18.8%	71%	445959	0.747
doMatc	23.856 ms	[336.1.1]	[512.1.1]	17.8%	75.8%	550455	0.714

⚠ Low Global Memory Load Efficiency

[kernels accounting for 100% of compute have low efficiency (13.3% avg)]

⚠ Low Global Memory Store Efficiency

[kernels accounting for 93.9% of compute have low efficiency (43.3% avg)]

⚠ Low Warp Execution Efficiency

[kernels accounting for 85.3% of compute have low efficiency (61.9% avg)]

Figure 5.5: A screen capture from `Nvidia nvpp` showing various metrics applied to our algorithm.

5.4 Summary

In this chapter we showed the results we obtained for our multicore and GPU algorithms.

For multicores we developed an efficient sharing mechanism that lead us to near-linear speedup for two different algorithms that use g-tries at their core. This paves the way for the usage of subgraph counting algorithms on larger networks and for bigger subgraph sizes on the user's personal multicore machine.

We also developed an initial approach to subgraph counting using GPUs. Much work has yet to be done in the field but there is much potential for this approach and interesting results can, in principle, be obtained.

Conclusions and Future Work 6

Complex Networks are used in virtually every field of study, with large real-world datasets being widely available. Finding patterns in their structure can lead to a better comprehension of their function. Specific patterns, called network motifs, have been extensively applied to biological networks but also to chemical and engineering networks. Building on previous fast algorithms for subgraph census, the purpose of this work was to develop parallel strategies that further speed up the finding of network patterns, increasing the size of the patterns that can be found in a reasonable amount of time and also in bigger networks.

This final chapter summarizes our main contributions and concludes with directions for future research.

6.1 Main Contributions

In this work we targeted both the multicore and the GPU architectures. The two of them are ubiquitous, being present on most of the personal computers today. This gives our work a large field of applicability. We now describe our main contributions for each of these approaches.

- **Multicore Approach:** Our implementation was done using `Pthreads` and the results were obtained with one thread per core. `Pthreads` are supported by all major operating systems, not limiting our work to a specific architecture.
  - We **developed two efficient parallel algorithms** to count subgraph frequencies for multicore architectures. They were based on two of the fastest algorithms for subgraph counting. Both used the g-trie data structure to encapsulate isomorphism information. G-Tries are multiway trees, much like prefix trees, that use common topologies in subgraphs in order to prune the search tree. The sequential versions of `gtrieScanner` and `FaSE`

  already performed significantly better than competing algorithms, making them a solid base for improvement. We were able to keep the original recursive nature of the counting algorithms only creating a more explicit work tree when needed. To dynamically divide the search tree among the threads, we developed an efficient sharing mechanism that is able to stop, split and resume the execution. By being able to successfully apply our sharing strategy to two different algorithms we also display our strategy's generality.

  – We performed a **thorough study of our algorithms' scalability** on several representative networks from various fields and presented near-linear speedup up to 32 cores. To the best of our knowledge, our parallel algorithms constitute the two fastest available methods for shared memory environments and allow practitioners to take advantage of either their personal multicore machines or more dedicated computing resources. This expands the limits of subgraph counting applicability, allowing an exploration of larger subgraphs in bigger networks.

- **GPU Approach:** We provided an initial algorithm built using the g-trie data structure. We studied the GPU architecture, and the `CUDA` model in particular, and identified the difficulties in adapting graph traversing algorithms to it.

6.2 Future Work

Much work remains to be done in the field, either by improving the algorithms executing times or by applying them to real world data and extract valuable information about the network's structure. Next we give a few points for the future.

- **Scalable GPU Approach:** In this work we presented an initial approach using GPUs. While it did not present very good results we intend to further explore this architecture and develop a more efficient algorithm. For this purpose we might have to put g-tries aside and find a strategy that more efficiently takes advantage of the GPU organization. Looking at the current best results achieved for breadth-first search in the literature [HKOO11, MGG12] it appears to be a challenging task but, at least, a reasonable speedup seems to be possible.

- **Mixing GPU and Multicore Approaches:** If a scalable GPU algorithm is achieved it would be interesting to combine it with our multicore approach. `CUDA` supports multi-threading in the way of *streams* but the management of those streams may not be trivial. Nevertheless, a strategy combining the two approaches would take full advantage of both the multicores and the GPU of a personal computer and could lead to very interesting results that would further expand the applicability of subgraph finding.

- **Mixing Distributed and Shared Memory Approaches:** Previous work has been done in distributed memory [RSL10a] and very promising results were obtained, with near-linear speedup up to 128 cores. Combining our multicore algorithm with a distributed approach could, in principle, obtain similar results. An initial idea could be to use distributed memory for communication between different machines from a cluster and use shared memory for the cores of each machine.

- **Study Real World Scenarios:** On a more practical angle, we may use our method to analyze several data sets, searching for new subgraph patterns that can lead to novel insight into the structure of these real-life networks. For example, we are in the process of building a large co-authorship network and plan to explore its structure using our algorithm.

6.3 Closing Remarks

The main objectives of this thesis were accomplished. Two parallel algorithms using a general work sharing mechanism were obtained and an initial study of the applicability of subgraph census to GPU computing was also made.

For the author, this work contributed to develop his programming skills, especially in the parallel programming paradigm. The scientific work produced, resulting in two accepted papers was most rewarding.

References

[AA04] I. Albert and R. Albert. Conserved network motifs allow protein-protein interaction prediction. *Bioinformatics*, 20(18):3346–3352, 2004.

[ABAU99] J Almunia, G Basterretxea, J Aristegui, and RE Ulanowicz. Benthic-pelagic switching in a coastal subtropical lagoon. *Estuarine, Coastal and Shelf Science*, 49(3):363–384, 1999.

[AFU13] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 62–73, Los Alamitos, CA, USA, 2013. IEEE CS.

[AG05] Lada A. Adamic and Natalie Glance. The political blogosphere and the 2004 u.s. election: Divided they blog. In *3rd International Workshop on Link Discovery*, LinkKDD '05, pages 36–43, New York, NY, USA, 2005. ACM.

[APR14] David Aparicio, Pedro Paredes, and Pedro Ribeiro. A scalable parallel approach for subgraph census computation. In *7th International Workshop on Multi/many-Core Computing Systems (MuCoCoS 2014)*. Springer, August 2014.

[Are14] Alex Arenas. Alex arenas datasets, 2014. http://deim.urv.cat/˜aarenas/data/welcome.htm.

[ARS14] David Aparicio, Pedro Ribeiro, and Fernando Silva. Parallel subgraph counting for multicore architectures. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE CS, August 2014.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

REFERENCES

[BB02] Christian Borgelt and Michael R Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 51–58. IEEE, 2002.

[BGP07] Kim Baskerville, Peter Grassberger, and Maya Paczuski. Graph animals, subgraph sampling, and motif search in large networks. *Physical Review E*, 76(3):036107, 2007.

[BM06] Vladimir Batagelj and Andrej Mrvar. Pajek datasets, 2006. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[BZC$^+$03] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, and Runsheng Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic Acids Research*, 31(9):2443–2450, 2003.

[CG08] G. Ciriello and C. Guerra. A review on models and algorithms for motif discovery in protein-protein interaction networks. *Brief Funct Genomic Proteomic*, 7(2):147–156, 2008.

[CRS12] Sarvenaz Choobdar, Pedro Ribeiro, and Fernando Silva. Motif mining in weighted networks. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, pages 210–217. IEEE, 2012.

[DA05] Jordi Duch and Alex Arenas. Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104, 2005.

[dFCRTB07] Luciano da F. Costa, Francisco A. Rodrigues, Gonzalo Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances In Physics*, 56:167, 2007.

[FFHV07] Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 340–351, 2007.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[GD03] Pablo M Gleiser and Leon Danon. Community structure in jazz. *Advances in complex systems*, 6(04):565–573, 2003.

[GK07] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. *Research in Computational Molecular Biology*, pages 92–106, 2007.

[Glä01] Wolfgang Glänzel. National characteristics in international scientific co-authorship relations. *Scientometrics*, 51(1):69–115, 2001.

[GS05] Wolfgang Glänzel and András Schubert. Analysing scientific networks through co-authorship. In *Handbook of quantitative science and technology research*, pages 257–276. Springer, 2005.

[HKOO11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *SIGPLAN Not.*, 46(8):267–276, February 2011.

[HN07] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[HWP03] Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552. IEEE, 2003.

[ILK$^+$05] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, and U. Alon. Coarse-graining and self-dissimilarity of complex networks. *Physical Review E*, 71(1 Pt 2), 2005.

[JCZ13] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Engineering Review*, 28(1):75–105, 2013.

[JHK12] Jeannette Janssen, Matt Hurshman, and Nauzer Kalyaniwalla. Model selection for social networks using graphlets. *Internet Mathematics*, 8(4):338–363, 2012.

[JJCL06] Ronald Jackups Jr, Sarah Cheng, and Jie Liang. Sequence motifs and antimotifs in β-barrel membrane proteins from a genome-wide analysis: the ala-tyr dichotomy and chaperone binding motifs. *Journal of molecular biology*, 363(2):611–623, 2006.

[JMA07] Ruoming Jin, Scott McCallen, and Eivind Almaas. Trend motif: A graph mining approach for analysis of dynamic complex networks. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 541–546. IEEE, 2007.

[KA05] Nadav Kashtan and Uri Alon. Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences of the United States of America*, 102(39):13773–13778, 2005.

[KAE$^+$09] Z. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC bioinformatics*, 10(1):318, 2009.

[KIMA04] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.

[Kon08] Michio Kondoh. Building trophic modules into a persistent food web. *Proceedings of the National Academy of Sciences*, 105(43):16631–16635, 2008.

[KRSA11] Varun Krishna, NNR Ranga Suri, and G Athithan. A comparative survey of algorithms for frequent subgraph discovery. *Current Science (00113891)*, 100(2), 2011.

[LBY$^+$04] Nicholas M Luscombe, M Madan Babu, Haiyuan Yu, Michael Snyder, Sarah A Teichmann, and Mark Gerstein. Genomic analysis of regulatory network dynamics reveals large topological changes. *Nature*, 431(7006):308–312, 2004.

[Les14] Jure Leskovec. Snap: Network datasets, 2014. http://snap.stanford.edu/data/.

[LFS06] Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):360–368, 2006.

[LKF05] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In

11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, pages 177–187, New York, NY, USA, 2005. ACM.

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[LLDM09] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[LRR+02] Tong Ihn Lee, Nicola J Rinaldi, François Robert, Duncan T Odom, Ziv Bar-Joseph, Georg K Gerber, Nancy M Hannett, Christopher T Harbison, Craig M Thompson, Itamar Simon, et al. Transcriptional regulatory networks in saccharomyces cerevisiae. *Science*, 298(5594):799–804, 2002.

[MGG12] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.

[MIK+04] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, March 2004.

[MKD+04] Hong-Wu Ma, Bharani Kumar, Uta Ditges, Florian Gunzer, Jan Buer, and An-Ping Zeng. An extended transcriptional regulatory network of escherichia coli and analysis of its hierarchical structure and network motifs. *Nucleic acids research*, 32(22):6643–6649, 2004.

[ML12] Julian McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In P. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 548–556. 2012.

[MP08] Tijana Milenković and Nataša Pržulj. Uncovering biological network function via graphlet degree signatures. *Cancer informatics*, (6), 2008.

[MSOI+02] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[New06] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3):036104, 2006.

[New10] Mark Newman. Network data sets, 2010. http://www-personal.umich.edu/˜mejn/netdata/.

[NK04] Siegfried Nijssen and Joost N Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652. ACM, 2004.

[NLGC02] Kim Norlen, Gabriel Lucas, Michael Gebbie, and John Chuang. EVA: Extraction, Visualization and Analysis of the Telecommunications and Media Ownership Network. In *International Telecommunications Society 14th Biennial Conference (ITS2002), Seoul Korea*, 2002.

[OSKK05] Jukka-Pekka Onnela, Jari Saramäki, János Kertész, and Kimmo Kaski. Intensity and coherence of motifs in weighted complex networks. *Physical Review E*, 71(6):065103, 2005.

[PCJ04] N Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[PCJ06] N Pržulj, Derek G Corneil, and Igor Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006.

[PIL05] Robert J Prill, Pablo A Iglesias, and Andre Levchenko. Dynamic properties of network motifs contribute to biological network organization. *PLoS biology*, 3(11):e343, 2005.

[PR13] Pedro Paredes and Pedro Ribeiro. Towards a faster network-centric subgraph census. In *International Conference on Advances in Social Networks Analysis and Mining*, pages 264–271. IEEE, 2013.

[Prž07] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[RS10] Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *ACM Symposium on Applied Computing*, 2010.

[RS14a] Pedro Ribeiro and Fernando Silva. Discovering colored network motifs. In *Proceedings of the 5th International Workshop on Complex Networks (CompleNet)*, March 2014.

[RS14b] Pedro Ribeiro and Fernando Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28:337–377, March 2014.

[RSL10a] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Efficient parallel subgraph counting using g-tries. In *IEEE International Conference on Cluster Computing (Cluster)*, pages 1559–1566. IEEE Computer Society Press, September 2010.

[RSL10b] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Parallel calculation of subgraph census in biological networks. In *1st International Conference on Bioinformatics*, Valencia, Spain, 2010.

[RSL12] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Parallel discovery of network motifs. *Journal of Parallel and Distributed Computing*, 72:144–154, 2012.

[San02] Peter Sanders. A detailed analysis of random polling dynamic loadbalancing. In *International Symposium on Parallel Architectures Algorithms and Networks*, pages 382–389. IEEE, 2002.

[SK04] Olaf Sporns and Rolf Kotter. Motifs in brain networks. *PLoS Biology*, 2, 2004.

[SM13] George M. Slota and Kamesh Madduri. Fast approximate subgraph counting and enumeration. In *42nd International Conference on Parallel Processing (ICPP)*, pages 210–219, 2013.

[SOMMA02] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature Genetics*, 31(1):64–68, 2002.

[TL09] Lei Tang and Huan Liu. Relational learning via latent social dimensions. In *15th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD09)*, pages 817–826, 2009.

[TMP12] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.

[TZvO07] John Tsang, Jun Zhu, and Alexander van Oudenaarden. Microrna-mediated feedback and feedforward loops are recurrent network motifs in mammals. *Molecular cell*, 26(5):753–767, 2007.

[Uni14] Arizona State University. Social computing data repository at ASU, 2014. http://socialcomputing.asu.edu/pages/datasets.

[VS05] Sergi Valverde and Ricard V. Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2), 2005.

[WDY13] Fei Wang, Jianqiang Dong, and Bo Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In *Intelligent Data Engineering and Automated Learning–IDEAL 2013*, pages 342–349. Springer, 2013.

[WMFP05] Marc Wörlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. *A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston*. Springer, 2005.

[WR06] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[WSTB86] John G White, Eileen Southgate, J Nichol Thomson, and Sydney Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 314(1165):1–340, 1986.

[WTZ$^+$05] Tie Wang, Jeffrey W. Touchman, Weiyi Zhang, Edward B. Suh, and Guoliang Xue. A parallel algorithm for extracting transcription regulatory network motifs. *IEEE International Symposium on Bioinformatics and Bioengineering*, pages 193–200, 2005.

[YH02] Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[YLSK$^+$04] Esti Yeger-Lotem, Shmuel Sattath, Nadav Kashtan, Shalev Itzkovitz, Ron Milo, Ron Y Pinter, Uri Alon, and Hanah Margalit. Network motifs in integrated cellular networks of transcription–regulation and protein–protein interaction. *Proceedings of the National Academy of Sciences of the United States of America*, 101(16):5934–5939, 2004.

[ZKKM10] Zhao Zhao, M. Khan, V. S. Anil Kumar, and M. V. Marathe. Subgraph Enumeration in Large Social Contact Networks Using Parallel Color Coding and Streaming. In *39th International Conference on Parallel Processing (ICPP)*, pages 594–603, 2010.

[ZWB$^+$12] Zhao Zhao, Guanying Wang, Ali R. Butt, Maleq Khan, V.S. Anil Kumar, and Madhav V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. *Parallel and Distributed Processing Symposium, International*, 0:390–401, 2012.