Università Ca'Foscari Venezia

Master's Degree

in Computer Science

Final Thesis

Modularity Based Community Detection on the GPU

Supervisor

Ch. Prof. Claudio Lucchese

Graduand

Federico Fontolan

Matriculation Number

854230

Academic Year

2019 / 2020

Abstract

Modularity based algorithms for the detection of communities are the de facto standard thanks to the fact that they offer the best compromise between efficiency and result. This is because these algorithms allow analyzing graphs much larger than those that can be analyzed with alternative techniques. Among these, the Louvain algorithm has become extremely popular due to its simplicity, efficiency and precision. In this thesis, we present an overview of community detection techniques and we propose two new parallel implementations of the Louvain algorithm written in CUDA and exploitable by Nvidia GPUs: the first one is based on the sort-reduce paradigm with a pruning approach on the input data; the second one is a new hash-based implementation. Experimental analysis conducted on 13 datasets of different sizes ranging from 15 to 130 million edges shows that the proposed algorithms have different efficiency based on the graph. For this reason, we study also an adaptive solution that try to improve the performance combining this two approaches.

Contents

1 Introduction

2 Nvidia GPUs architecture and CUDA

The scientists, years by years, have to face bigger and bigger problems. Even if Moore's law (that said: "the number of transistors in an integrated circuit double about every two years") determines the increased power of the CPU since the sixties, even the problems size grows and it grows also much more quickly. For example, the web growth since the turn of the millennium raises new challenges that are hard to solve with the standard algorithm, due to the size of the data.

Besides, at the same time, the manufacturers have to face some serious physical limits. The increase in performance was made possible by the reduction in the size of the transistors and the increase in the frequency of the clock cycle. In the first half of the two-thousands, the producers discovered that reducing, even more, the size cause serious problems of heat dissipation and data synchronization. To find a solution to this problems, the manufactures start to produce multi-core CPU: the idea is that if it's impossible to increase further the speed with only one core, they add another processing unit, to ideally halve the execution time.

For these reasons, in recent times the studies of new parallel approaches became fundamental to solve problems that can not be solved classically. As a result of this change and at the same time both the support of floating-point number on the graphics processing units (GPU) and the advent of programmable shaders, it became popular the general-purpose computing on GPU (GPGPU), i.e. the use of a GPU to perform a computation that is commonly handled by the CPU.

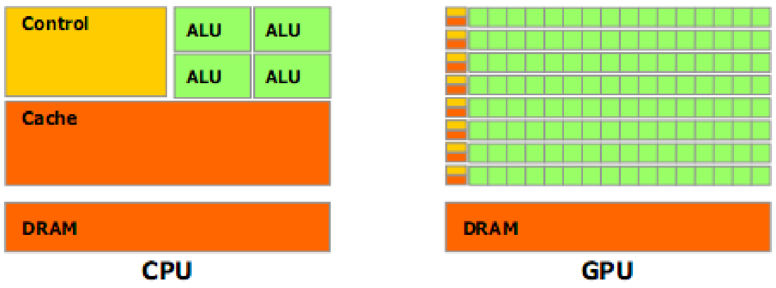


Figure 1: Difference in CPUs and GPUs architecture. This image was reprinted from [25]

The GPU architecture is radically different respect to the CPU. The CPU has several ALUs (Arithmetic and Logic Unit), a complex control unit that controls those ALUs, big fast cache memory and dynamic random access memory (DRAM); the GPU has many ALUs, several simple control units, a smaller cache and a DRAM (Figure 1). While the first one is focused on the low-latency, the second one is focused on the high throughput; while the first one is focused on handle various serial complex instruction, the second is focused on handle much parallel simple instruction. In brief, the first one is a versatile processing unit, the second one is highly specialized. Even if in recent time the multi-core CPU performance get closer to the performance of the GPU [27], from the Figure 2 we can see how the performance of the GPU outclasses the performance on the CPU.

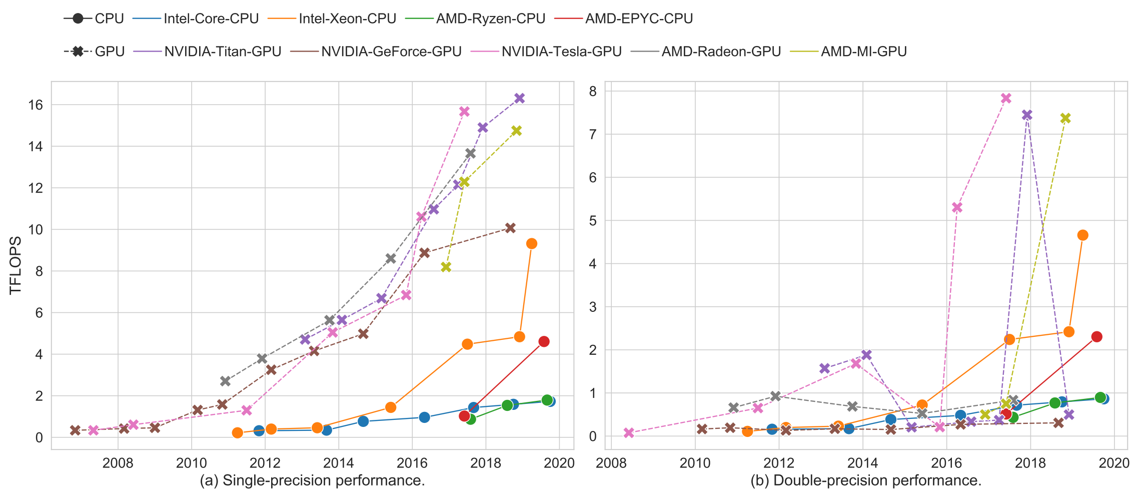
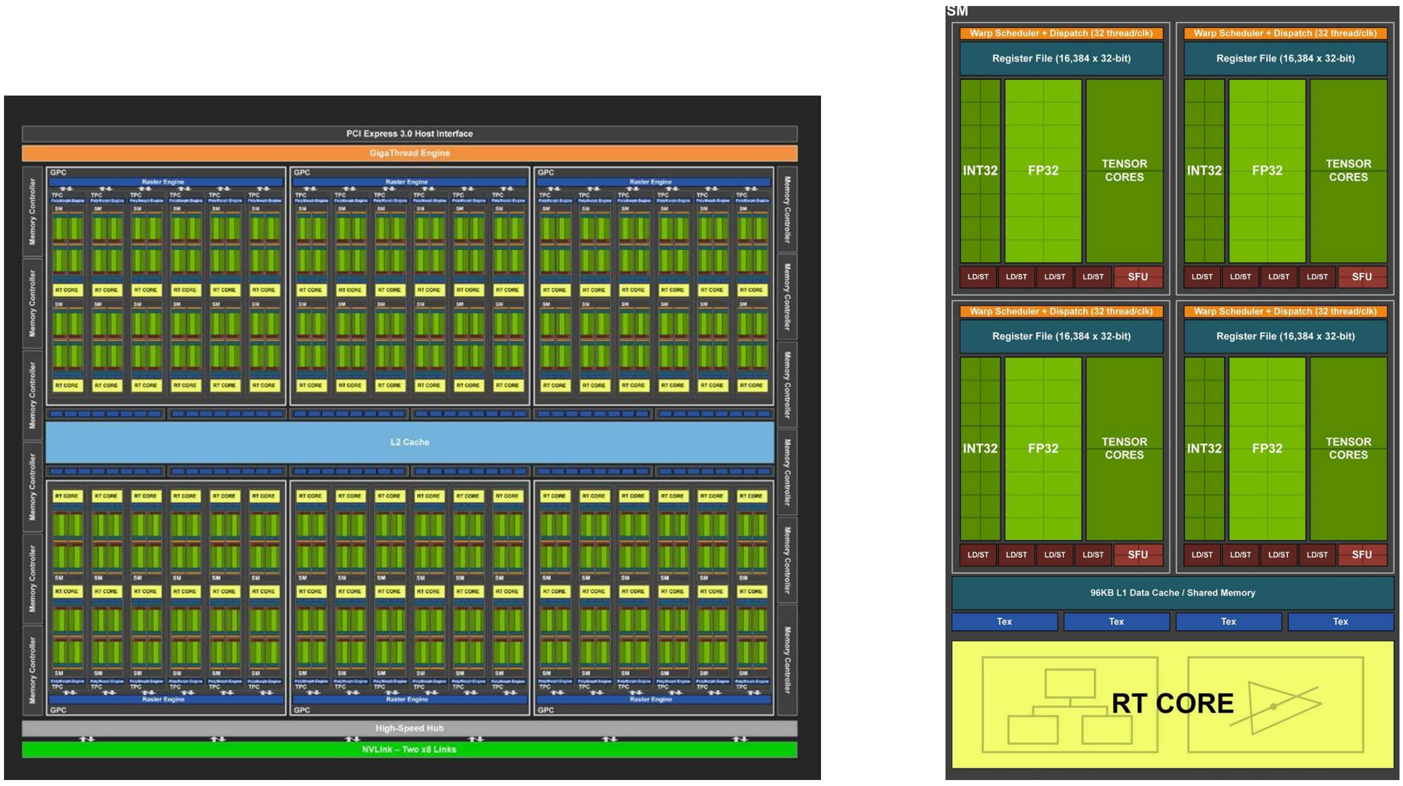


Figure 2: Comparing single-precision and double-precision performance of CPUs and GPUs. The performance are measured in trillion of floating point operations per Second (TFLOPS). This image was reprinted from [27].

For those reasons, the GPU-accelerated applications are the most effective to solve big problems, due to the possibility of reach very high speed-up compared to the classic multi-core applications. To simplify the development of this type of applications, in 2007, Nvidia releases CUDA (Compute Unified Device Architecture), a parallel computing platform and application programming interface (API) model. Nowadays, the CUDA framework is one of the main tools to develop HPC applications, due to its performance and simple API, and for this, we choose to use it in this project. In this chapter, first of all, we present the Nvidia's GPU architecture, then after we present CUDA basis and Thrust, a library that was used in the project presented in this thesis that simplifies the development. This chapter was based on [26] and [25].

2.1 Nvidia's GPU Architecture

Now we present Nvidia's GPU Architecture to introduce some key concept that is used later in the thesis. This introduction present the last architecture released by Nvidia, Turing. We present the highest performing GPU of the Turing line, The Turing TU102 GPU (Turing machine can be also scaled down to the scaled-down from this one). In Figure (3a) we can see a scheme of this architecture. The cornerstone of each Nvidia's GPU is the concept of Streaming Multiprocessor



(a)

(b)

Figure 3: (a): Tuning GPU full architecture; (b) Streaming multiprocessor (SM) in details. Those images was reprinted from [26]

(SM), that are represented in Figure (3b): it contains some cores specialised to solve specific arithmetic operations on specific types of data (like integer, float, double, tensor...). In a Tuning machine, each SM contains 64 FP32 cores, 64 INT32 cores,

eight Tensor cores and two FP64 cores (that aren't present in Figure 3b). In tuning architecture is present also a Ray Tracing cores in each SMs: this core is used in rendering.
The SM is the fundamental unit because, as we see soon, the parallel execution of the code in a CUDA application it's organized in blocks, and each block is executed on a single SM. Moreover, the SM contains also some register (256 KB in Turing), an L1 cache and a shared memory (in Turing 96 KB of L1/shared memory which can be configured for various capacities). The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps: when a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. A very important notion is that each warp executes one common instruction at a time, so if threads of a warp diverge via a conditional branch, the warp serially executes each branch path taken, ignoring the instruction for the threads that are not on the active path. The registers are private for each thread, but all threads share the SM's shared memory. The SMs are organised in Texture Processing Clusters (TPCs), that in a Turing GPU contains two SMs. In their turn, the TPCs are organized in Graphics Processing Clusters (GPC) that in a TU102 contains six TPCs. Finally, each GPU's contain six GPCs. Shared to all component there is an L2 cache: this is used as global memory, i.e. each thread can have access to it. In the Turing GPU. it is large 6144 KB. Therefore, in summary, a Turing TU102 GPU contains 72 SMs and than 4608 FP32 cores, 4608 INT32 cores, 576 tensor core and 144 FP64 cores.

2.2 CUDA

In November 2006, CUDA (that stands for Compute Unified Device Architecture) was realised by NVIDIA. This general-purpose parallel computing platform aims to give a framework to the developers that allow building applications that transparently scale its parallelism with a low learning curve. To overcome this challenge, CUDA was designed as a C++ language extension: in this way, a programmer that already know the language syntax can start to develop a GPU-accelerated application with a minimal effort. The support of other language was introduced years by

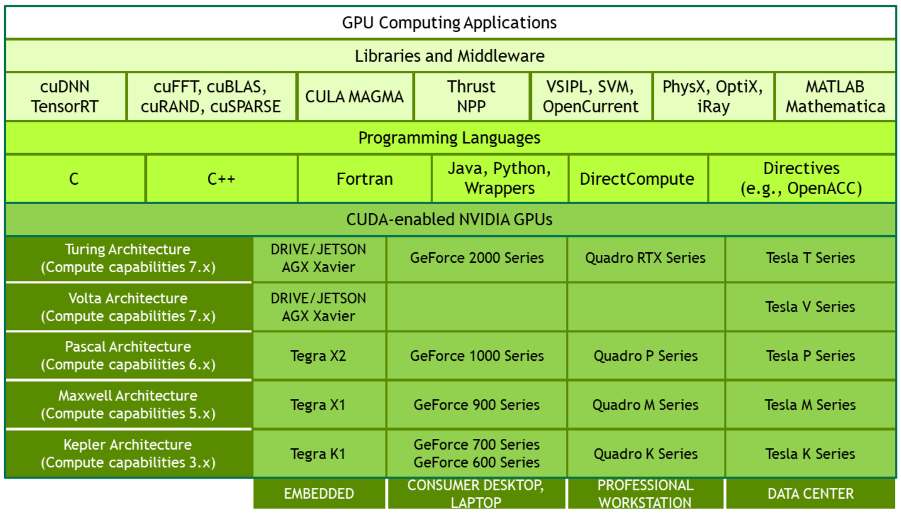
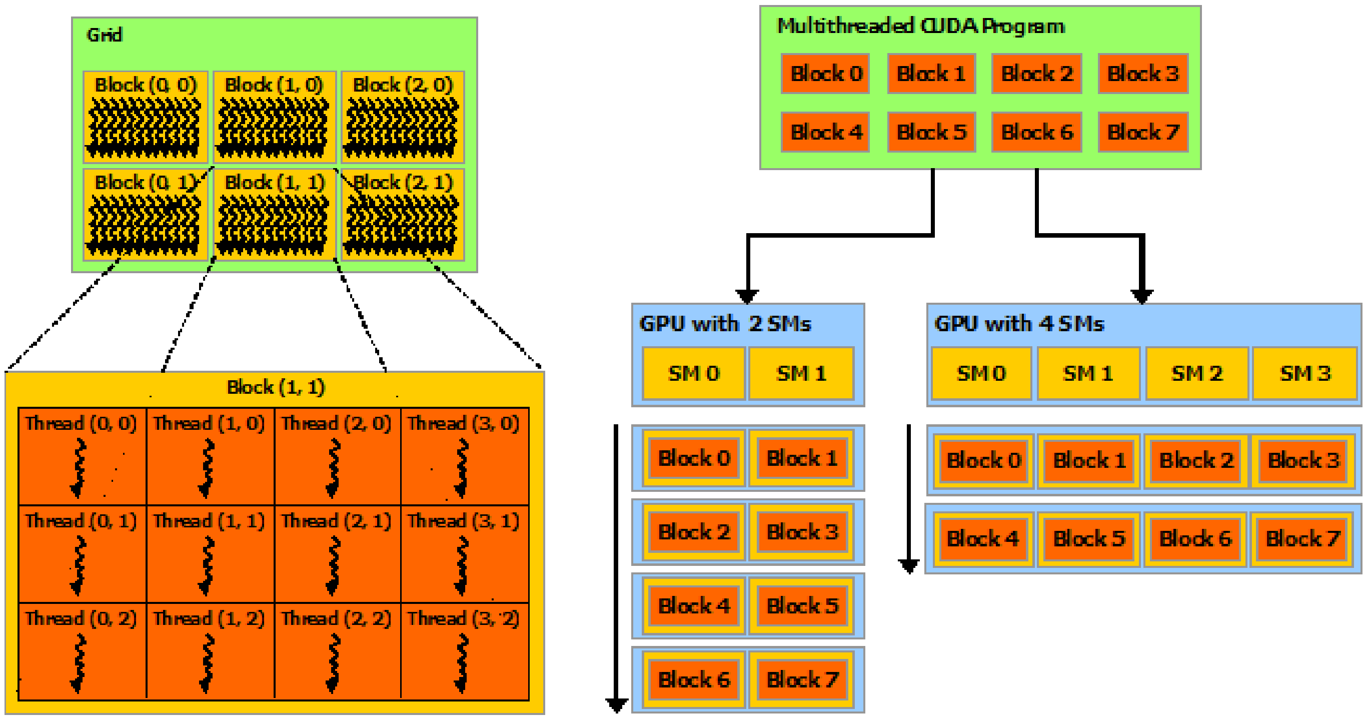


Figure 4: GPU Computing Applications. This image was reprinted from [25].

years, as illustrated in Figure 4. In this chapter we present the C++ extension, that was used to develop the project illustrated in this thesis, even if all extensions share the same concept and programming model.

The first key concept is the **kernel** function. In a CUDA-based application we define as **device** the GPU and as **host** the CPU. The application starts to the host, and when it is needed, it calls a kernel function that executes the function N times in parallel by N different threads. To define a kernel, we have to add `__global__` declaration specifier to the method and the number of thread that have to execute the kernel call. Each thread has a unique ID. To set this number we use an execution configuration syntax: after the method name, we include this setup enclosed in three angle brackets `<<< ... >>>`. The configuration is used to define the number and sizes of the blocks: a block is a group of threads that are organized in a one, two or three dimensional way. To identify the threads referring to the block, they have the three-component vector `threadIdx` that identify its position. In their turn, also the blocks are organized into a one-dimensional, two-dimensional or three-dimensional grid. Similar to the previous one, the vector `blockIdx` identify the block into the grid, and each thread can see each owns. To define the dimensions of the grids and the blocks in the angle brackets, we use two `dim3` (or eventually `int` to define a one dimension grid/blocks). The total number of threads is equal to the number of threads per block times the number of blocks: using the same logic, we can recover



(a) (b)

Figure 5: (a): Grid of Thread Blocks; (b) Automatic Scalability. Those images was reprinted from [25]

the unique ID of the vector from `threadIdx` and `blockIdx`. In Figure 5a is illustrated the grid-blocks schema.

As mentioned above, each block is assigned to a different streaming multiprocessor. On current GPUs, a block has threads limit set to 1024, due to the limited memory resources of the SM. This block scheme is used to implement automatic scalability: indeed, the GPU schedule each block thread on any available SM, in any order. For example, if we have a program that divides the threads into eight blocks, it can be executed from both two GPU's with respectively two and four SMs without any intervention on the scheduling from the developer (Figure 5b). By other hands, the block schema allows also threads collaboration: as illustrated in the Figure 6, thanks to the allocation of each block to the same SM, allows those threads to share a fast per-blocks memory. Besides, all threads share the global device memory, even if they belong to different blocks or kernel (some advanced settings permit to execute two kernels simultaneously if there are enough resources. Those settings are not presented here because there aren't used in this thesis because a large amount of

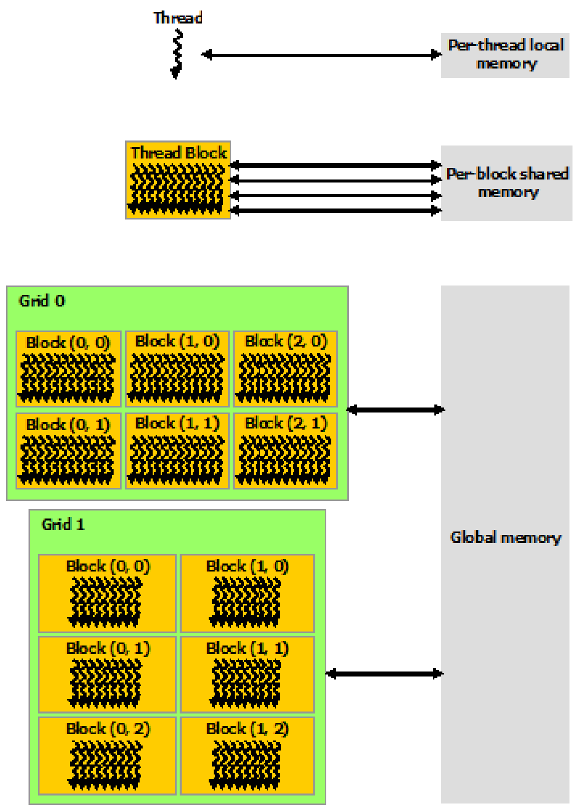


Figure 6: Memory Hierarchy. This image was reprinted from [25].

data doesn't permit to parallelize those kernels. Moreover, for the sake of completeness, they are well described in [25]). The data must be copied to this memory from the host before the kernel execution. Those two type of memory, combined to several primitives that synchronize thread at warp, blocks or device levels, permitting threads collaboration. Those synchronizing function acting as a barrier: all threads in the specific level must wait for the others before any is allowed to proceed. In addition, CUDA exposes some other primitives that allow atomic operations: if multiple threads call one of those methods on a specific memory address, the access to it will be serialized. No information about the order of the operation will be given a priori. In conclusion, we remark that every new hardware architecture could introduce new features that aren't supported by the old GPU. For this reason, CUDA uses the concept of Compute Capability to identify the features supported by the GPU hardware.

2.3 Thrust Library

To conclude this CUDA introduction, we present Thrust, a powerful library of parallel algorithms and data structures that are largely used in this thesis project. This C++ Standard Template-based library is included in the CUDA toolkit and provides a reach collection data-parallel primitives (as transform, sort or reduce) that allows writing a high performing and readable code with minimal effort. This presentation is based on the Thrust section present in the CUDA manual [25].
We start the presentation from the two vector containers, `host_vector` and `device_vector`. As their name says, they are arrays that are dynamically allocated respectively in the host and the device memory. Like the `std::vector`, they are generic containers, their elements are allocated in contiguous storage locations and they can dynamically change the size. Indeed, using the `=` operator, we can copy a `host_vector` in a `device_vector` and vice-versa. Thrust also provides many useful parallel algorithms, implemented for both host and device, like:

- Sort that performs the sorting of a vector. It is also present its "by key" version that sorts a vector of values using another vector as a key;
- Reduce that performs a reduction of a vector. It is also present its "by key" version that, given a vector of values and a vector of keys, performs a reduction of the values for each consecutive group of keys;
- Transform that applies a function to each element of the vector;
- Exclusive and Inclusive Scans that perform a prefix sum, respectively ignoring and considering the corresponding input operand in the partial sum.

In this thesis, we use only the device CUDA-based version of this algorithm. The last useful feature that Thrust provides is the fancy iterators. These iterators are used to improve performance in various situations. The `transform_iterator`, for example, were used to optimize the code performing the transformation during the execution of an algorithm. Another very useful iterator is the `zip_iterator` that takes multiple input sequences and yields a sequence of tuples: in this way we can treat many vectors as a single one and perform more operations simultaneously.

3 Community Detection State of the Art

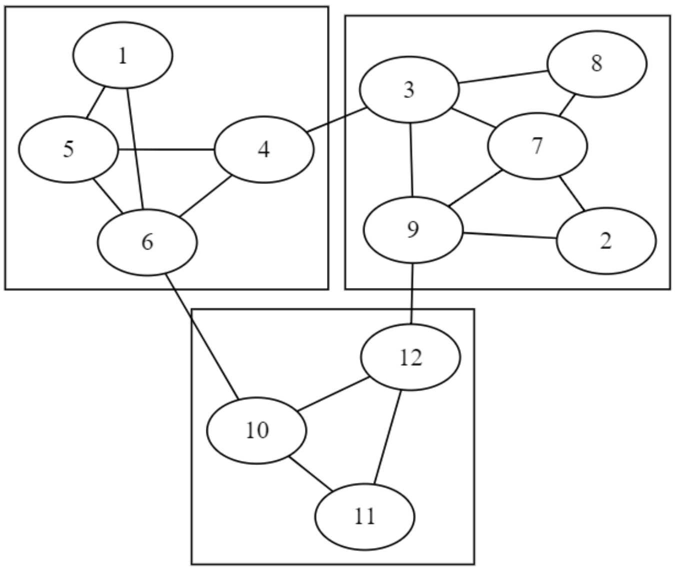


Figure 7: An example of a communities structured graph. In it is well visible three community, enclosed by the rectangles. The image was made with Graphviz.

The problem of community detection raises in many application scenarios from the necessity of finding groups of objects that have a large number of connections to each other. To represent problems where it is fundamental to empathize connection between objects, the graph theory is the main tool. A graph is a mathematical structure composed of nodes (or vertices) that denote the objects and edges (or links) that express some kind of relationship between objects and possibly having a weights that quantifies this relationship. The Graph Theory born in 1736 when Euler used this mathematical abstraction to solve the puzzle of Königsberg's bridges. Since then, this tool was used in several of Mathematics, Social, Biological and Technological application. In recent time, the approach to this studies has been revolutionized to deal with bigger and more complicated challenges, supported by the increasing computing power.

The necessity of finding this high-connected substructure in graph arises from real problems of the previous field: for example, the study of Protein-Protein Interaction (PPI) networks is very important because the interaction between proteins is the basis of all process in the cell.

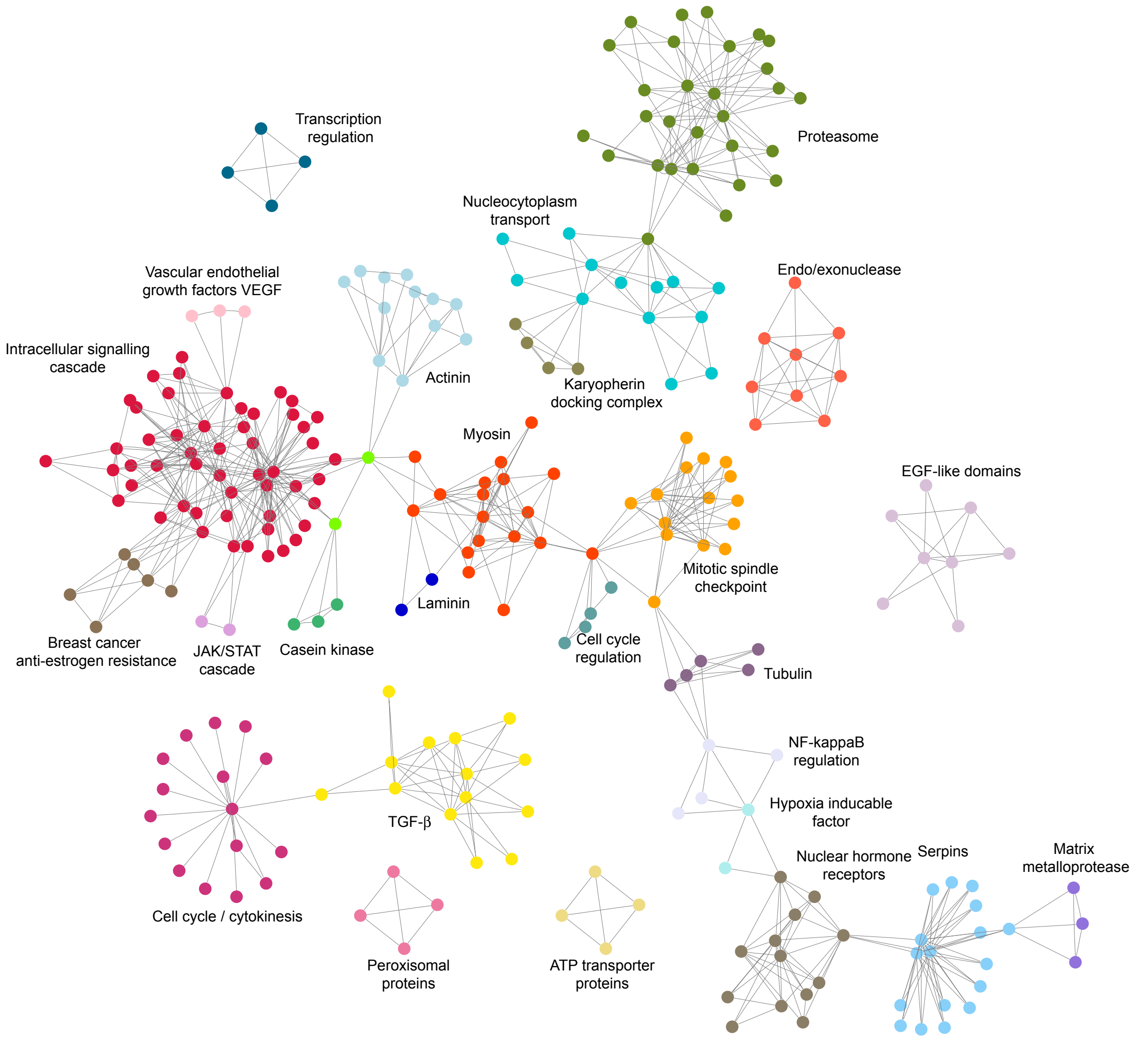


Figure 8: A protein protein iteration network of a rat cancerous cell. This image was reprinted from [12].

A study demonstrated that this type of network shown to be useful for highlighting key proteins involved in metastasis. [12]

Other examples can be found in the field of sociology: a historically well-know scenario is the Zachary's Karate Club. This dataset captures members of a Karate Club for 3 years.[4] An edge between two nodes represents an interaction between two members outside the club. At some point, a conflict between the administrator and a master led to split of the club into two separate groups. The question is if it is possible to infer who compose these two new groups basing on the information that this graph give to us. This small network of 1977 is famous because it has often been

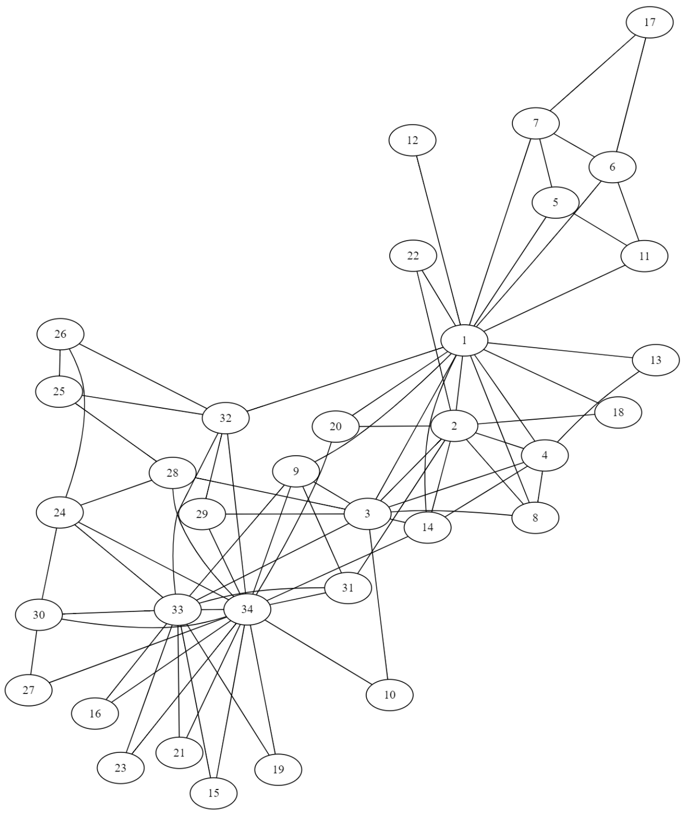


Figure 9: Zacahry's karate club. [4] This image was made with Graphviz.

used as a reference point to test the detection algorithms used to analyze huge social web networks. In general this kind of problem, i.e. clustering people that belong to the same community base on interaction, it's useful not only in sociology but also in marketing: by knowing people with similar interests, it's possible to make better recommendation systems.

There are several of similar scenarios to apply this method in the real-world, all united by the fact that the data are unregular but it's present some well-defined topological structure that in a completely random graph are absent. A random graph is a fully disordered graph, firstly proposed by Erdös and Rényi [2] in 1959: it's a graph where the probability that there is an edge between two nodes it's equal for all pairs of nodes and, for this reason, the degree of the nodes (i.e. the number of edges incident to a node) is homogeneous. In real networks, this is not true, because they are often scale-free (fallow a power-law distribution). An example of this is the study about the citations in scientific papers made by Derek J. de Solla Price in 1965 [3] or the study about World Wide Web growing made by Albert-László

Barabási et al in 1999 [5]. Furthermore, the degree distribution of the nodes is non-homogeneous not only globally but also locally, this due to the observation that there is a high concentration of edges within sets of nodes and a low concentration of edges between this sets. These two concepts are essential to formulate the formal definition of Community and Modularity. In this chapter will be presented some definitions of community and will be given an overview of some methods that are used to identify communities.

3.1 Community Definitions

The informal definition of community is there are many more edges inside the community versus the rest of the graph, but there isn't a unique quantitative definition of community. This kind of freedom is necessary because the concept of community is strictly connected to the problem that will be analyzed: for example, in some cases, it's necessary that community overlap, but in other problems, this is not necessary. There is a unique key constraint that allows talking about community detection: the graph must be sparse. A sparse graph is a graph where the number of nodes has the same magnitude of the number of edges. In the unweighted graph case, if the number of edges is far greater than the number of nodes, the distribution of edges among the nodes is too homogeneous for communities to make sense [15]. In that case, the problem nature is little different: we aren't interested anymore on the edge density between nodes but we have to use some kind of metrics (like similarity or distance) to clustering. In that case, the problem is more similar to data clustering. Despite this, assuming that a community is a subset of similar nodes it's reasonable, for this reasons some techniques (like spectral o hierarchical clustering) belonging to this field are adopted in community detection and will be shortly presented later on this thesis. Following this, Fortunato [15] defines three main classes of community's definitions: *local, global and based on vertex similarity.* Other types of definitions are still possible, but these three offers give a good summary of the problem. Now those classes will be presented to give an overview of the various approach that has been used to define this problem.

3.1.1 Local definitions

Considering that a community has a lot of interactions with the other nodes that are in it and few connections outside, it is fair to think about the communities as autonomous objects. The local definitions are based on this concept. Directly from this concept, we can think at the community as a clique, i.e. a subset whose vertices are all adjacent to each other. This type of definitions it's too strict: even if just one edge is not present, the subset is not a clique, but the subset has a very high concentration of edges. For this reason, the clique definition is often relaxed, using, for example, n-clique, i.e. a subset in which all the vertices are connected by a path of length less than n.
Anyway, this type of definitions ensure that there is a strong cohesion between the nodes in the subset, but not ensure that there isn't a comparable cohesion between the subset and the rest of the graph. For this purpose, other definitions were proposed. Given a graph $G(V, E)$, the relative adjacency matrix A and a subset of nodes C where $C \in V$, we define the internal degree k_v^{int} and the external degree k_v^{ext} for each vertex v that belongs to C as the number of edges that connect the node v with another node that belongs to C and not belongs to C, respectively:

$$k_v^{int} = \sum_{k \in C} A_{vk} \qquad\qquad k_v^{ext} = \sum_{k \notin C} A_{vk} \tag{1}$$

We also define the internal degree k_C^{int} and the external degree k_C^{ext} as the sum of all internal and external degree of nodes that belongs to C.

$$k_C^{int} = \sum_{i,j \in C} A_{ij} \qquad\qquad k_C^{ext} = \sum_{i \in C, j \notin C} A_{ij} \tag{2}$$

A strong community is a subset of nodes such that the internal degree k_n^{int} for each vertex n is greater than its external degree k_n^{ext} . This type of definitions once again very strict, for this reason we define as weak community a subset of nodes where the internal degree of the subset k_C^{int} is greater than its external degree k_C^{ext}. Many other variants of these definitions were presented in the literature.

3.1.2 Global definitions

The previous class quantify the community independently, considering every subset individually. Overturning the point of view, we can define communities in a graph-dependent way, considering them as an essential and discriminant part of it. There are many different interpretations of this approach in the literature, but the most important definitions are focused on this key fact: it's not expected to see a community structure in a random graph. For this reason, we define as *null model* of a graph another graph that have some features in common with the original one but it's generated randomly. This graph is used as a comparison term to identify if it's present a community structure in the graph or not and, if it is present, to quantify how it is pronounced. This approach, which is based the Modularity Optimization, is the main object of this study and is presented in detail in the next chapter.

3.1.3 Based on Vertex Similarity

The last class of definitions assumes that edges in the same community are similar to one another. All the definition used in the classic clustering methods belongs to this class because they calculate a distance (similarity) between object and aren't based on the edge density like the previous definitions. This distance can be calculated in various ways: if it is possible to embed the vertices into a n-dimensional Euclidean space by assigning a position to them, one method consists to calculate the distance between two nodes, considering that similar vertices are expected to be close to each other. To calculate the distance, one could use a norm. Three norms often used in the literature are the following. Given two points $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$ that belongs to the n-dimensional euclidian space E, we define the norms l_1 (Manhattan distance), l_2 (Euclidian distance) and l_3 (Maximum distance) as:

$$l_1(a,b) = \sum_{k=1}^{n} |a_k - b_k| \tag{3}$$

$$l_2(a,b) = \sum_{k=1}^{n} \sqrt{(a_k - b_k)^2} \tag{4}$$

$$l_3(a,b) = \max_{k \in [1,2]} |a_k - b_k| \tag{5}$$

Another option is the cosine similarity $\cos(a,b)$, that is very popular in literature:

$$\cos(a,b) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2}\sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{6}$$

If it is not possible to embed the graph in a Euclidean Space, it is possible to infer the distance from the adjacency matrix. If it is not possible to embed the graph in a Euclidean Space, it is possible to infer the distance from the adjacency matrix. One idea is to map the distance in order to assign smaller values at nodes with the same neighbourhood. Given an adjacency matrix A we define the distance between two nodes a and b as:

$$d(a,b) = \sqrt{\sum_{k \neq a,b} (A_{ak} - A_{bk})^2} \tag{7}$$

Many other variants of that definition (but based on the same principle) were presented in the literature, for example considering the overlap between neighbourhood respect to the union.

Other alternative measures consider the number of independent paths between nodes, i.e. path that does not share any common edges, or they are based on random walk on a graph: for example, the average number of steps needed to reach one vertex from another by a random walker.

3.2 Community Detection Algorithms

A partition is a division of the graph in clusters, such that each vertex belongs to exactly one cluster. The partition of possible partitions of a graph G with n vertices grows faster than exponentially with n, thus making it impossible to evaluate all the partitions of a graph [15]. For these reasons, many techniques were introduced to find the most significant ones. We now present an introduction to some classical class of techniques used in the field of community detection: Partitional clustering, Graph partitioning, Spectral clustering, Hierarchical clustering. Moreover, the Girvan and Newman algorithm is presented later on: even if this method is a Hierarchical algorithm, this method firstly introduced the modularity function and it is presented separately. The goal of this chapter is to give a useful overview in order to get the differences with the Modularity optimization and empathize the motivations that led to the definition of the Louvain algorithm, one of the most used nowadays, especially for huge graphs. For this reason, all the methods that are presented in this thesis find a partition, as the Louvain methods. For the sake of completeness, we remark that in Fortunato's report [15], that was mainly used to write this chapter, is presented an analysis of algorithms that found also overlapping communities (covers).

3.2.1 Partitional clustering

Partitional clustering is a class of methods that find clusters from data points. The algorithms in this class embed the graph in a metric space as seen in chapter 3.1.1, and then calculate the distance between these new points. The goal is to separate the points in k clusters minimizing the distance between points and to the assigned centroids (i.e. the arithmetic mean position of all the points in the cluster). The number of clusters k is given as input. The most famous technique is k-means clustering. The objective function to minimize is the following:

$$\sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - c_i||^2 \tag{8}$$

where C_i is the i-th cluster and c_i is its centroid. This function quantifies the intracluster distance. At the start, the k centroids are set far distance from each other. Then, each vertex is assigned to cluster with the nearest centroid and the centroid is recalculated. Even if the method doesn't find an optimal solution and the solution is strongly dependent on the initial setup of the centroids, this method remains popular due to the quick convergence that allows it to analyze big graphs. However, setting the apriori number of cluster k is not simple to estimate that number, especially in a large graph, and for this reason, it is often preferred algorithms that can automatically derive it. Moreover, the embedding of the graph in the Euclidean Space may be tricky and not reliable for some graphs.

3.2.2 Graph partitioning

Given a graph $G(V, E)$ and a number g of clusters, the problem of graph partitioning consists of creating a partition of nodes composed by g subset such that it minimizes the edges that lying between the clusters. To archive this goal, many algorithms performs a bisection of the graph, even for partitions with more than two clusters, where the bisection is iterated. One of the earliest and famous algorithms is the Kernighan–Lin algorithm. This algorithm performs an optimization of the function $Q = link_{in} - link_{between}$, where $link_{in}$ is the number of edges inside the subsets and $link_{between}$ is the number of edges lying between them. The algorithm starts from an initial partition (randomized or suggested by the graph), and the algorithm performs a swapping between clusters for a fixed number of nodes pair to increase the value of Q. To avoid local maxima, some swaps that decrease Q are kept. With some optimizations, the complexity of this algorithm is $O(n^2)$ where n is the number of nodes.
Other techniques are based on the max-flow min-cut theorem by Ford and Fulkerson [1] and the minimization of cut-affine measures, like the normalize cut:

$$\Phi_N(C) = \frac{c(C, V/C)}{k_c} \tag{9}$$

where C is a subset of nodes and k_c is the total degree of C.
Like the previous class, specifying the number of clusters is the greatest limit of this class of algorithm. In additions, iterative bisecting can lead to not reliable clusters, because the sub-clusters are made breaking the previous ones: in this way, the new subsets have vertices only from one of the "parent" cluster.

3.2.3 Spectral clustering

Given a set of n object $x_1, x_2, ..., x_n$ and the matrix S of pairwise similarity function $s(x_1, x_2)$ such that s is symmetric and non negative, we define as spectral clustering all methods that clustering all this object using the eigenvector derived from the matrix S. In particular, this transformation makes a change from the reference system of the object to another whose coordinates are elements of eigenvectors. This transformation is made to enhance the proprieties of the initial data. After that, we can cluster the data using other techniques as k-means and obtain a better result. The Laplacian matrix is the most used in spectral clustering. Given a graph G and its associated adjacently matrix A, we define the Laplacian matrix L of the graph G as:

$$L = D - A \tag{10}$$

where D is the degree matrix, a diagonal matrix which contains information about the degree of each vertex. This matrix is used due to nice propriety: if the graph has k connected components, the Laplacian of the graph will have k zero eigenvalues. In that case, the matrix can be organized in a way that displays l square blocks along the diagonal. When is it in this block-diagonal form, each block is at his turn a Laplacian matrix of one of the subcomponent. There are in this situation k degenerate eigenvectors with equal non-vanishing components in correspondence with the vertices of a block and zero otherwise. Considering the $n \times k$ matrix where n is the number of nodes of G and the columns of this matrix are the k eigenvectors, we can see that vertices in the same connected component of the graph coincide. If the graph is connected but the connections between the k subgraph are weak, only one eigenvalue is zero. By the way, However, the lowest $k - 1$ non-vanishing eigenvalues are still close to zero and the vertex vector of the first k eigenvectors still identify the clusters.

An application of these techniques is the spectral bisection methods: this algorithm combines ideas from spectral clustering and graph partitioning. Given the graph G

with n nodes, the cut size R of the bipartition of the graph is:

$$R = \frac{1}{4} s^T L s \tag{11}$$

where L is the Laplacian matrix and s is the n-vector that represents the affiliation of the nodes to a group (if the node i belongs to the first group, the i-th entry of s will be 1, -1 otherwise). s can be writtens as $s = \sum_{i=0}^{n} a_i v_i$ where v_i is the i-th eigenvector of the Laplacian. If s is normalized, we can write the equation (12) as the following:

$$R = \sum_{i=0}^{n} a_i^2 \lambda_i \tag{12}$$

where λ_i is the eigenvalue corresponding to v_i. From this, choosing the s parallel to the second-lowest eigenvector λ_2 we have a good approximation of the minimum because this would reduce the sum to λ_2. We remark that we use the second one because the first one is equal to zero. To cluster the data in the vector s, we match the signs of the components of v.
The exact computation of the all eigenvalues required a time $O(n^3)$, a too high complexity for big graph, but exist some techniques that allow calculating approximate values faster [15].

3.2.4 Hierarchical clustering

The possible partitions of a graph can be very different in scale: some cluster at its turn may show an internal community structure. In that case, there is a hierarchy between partitions. The most common ways to represent this kind of structure is to draw a dendrogram, i.e. a diagram representing a tree. The hierarchical clustering algorithms build an entire dendrogram starting or from the bottom (agglomerative algorithms), or from the top (divisive algorithms) using a similarity function to cluster. In the first type of algorithms, each node starts in its communities and the clusters are iteratively merged if the similarity score exceeds a threshold. By the other hands, a divisive algorithm overturns the starting point: at the start, all nodes belong to one singleton community and the clusters are iteratively split. An example of this type of algorithm, the Girvan and Newman algorithm, will be presented later on this thesis. Respect to the previous, the algorithms that belong to this class doesn't need the number of clusters as input, but there is the problem of discriminating between the obtained partitions. The Modularity Function was introduced to overcome this problem. Moreover, as we see in the Girvan and Newman algorithm, building the entire hierarchies using similarity metrics required a lot of computations: for these reasons the complexity of this class of algorithms tends to become much heavier if the calculation of the chosen similarity measure is costly [15].

3.3 Modularity Optimization

Historically, the modularity function Q was introduced as a stop criterion for the Girvan and Newman algorithm in 2002. This is a quality function, i.e. a function that allows distinguishing from a "good" cluster and a "bad" one. The function assigns to a partition a score that is used to compare partitions. This is not a trivial goal, because define if a partition is better than another is an ill-posed question: the answer may depend on the particular concept of community that it is adopted. Nevertheless, this sometimes is necessary, for example in the case of hierarchical clustering, where it's necessary to identify the best partition in the hierarchies. A simple example of this kind of function are the sum of the difference between internal degree k_v^{int} and the external degree k_v^{ext} [3.1.1].
The modularity function became very popular and a lot of methods based on this quality function were created. In this chapter we present the functions and its limits in details and the algorithm in which it was firstly used, some optimization techniques based on modularity.

3.3.1 Function

The function is based on the idea that we did not expect to see a graph structure in a random graph. We define as a *null-model* of a graph another one that it's generated randomly keeping some structural proprieties of the original one. Comparing the graph with its null model, we can quantify how much is well defined the community structure. Therefore, the modularity function is dependent on the choice of the null model. Given an undirected graph $G = (V, E)$, a partition of nodes C and a function $c(x)$ that assign each nodes x to its community, we define a generic modularity function as :

$$Q = \frac{1}{2|E|} \sum_{i,j \in V} (A_{ij} - P_{ij}) \delta(c(i), c(j)) \tag{13}$$

where A is the adjacency matrix of G, P is the matrix of expected number of edges between nodes in the null model and δ is an filter function: its yields one if $c(i) = c(j)$, zero otherwise.

In principle, the choice of a null model is arbitrary, but we have to consider carefully the graph properties to keep in the null model because they determine if the comparison is fair or not. For instance, it's possible to choose as a model that keeps only the nodes and edges numbers, assuming that an edge is present with the same probability for each pair of nodes (in this case P_{ij} is constant). For this reason, The standard null model of modularity imposes that the expected degree sequence(after averaging over all possible configurations of the model) matches the actual degree sequence of the graph [15]. In this scenario, the probability that two vertices i and j are connected by an edge is equals to the probability to get two stubs (i.e. half-edges) incident to i and j.

This probability p_i of piking a stub from the nodes i is $\frac{k_i}{2|E|}$ where k_i is the degree of nodes i. The probability that two stub joining is $p_i p_j = \frac{k_i k_j}{4|E|^2}$. Therefore, the expected number P_ij of connections between the nodes i and j is:

$$P_{ij} = 2m p_i p_j = \frac{k_i k_j}{2|E|} \tag{14}$$

Replacing P_{ij} from (14) in (13) we obtain:

$$Q = \frac{1}{2|E|} \sum_{i,j \in V} \left(A_{ij} - \frac{k_i k_j}{2|E|} \right) \delta(c(i), c(j)) \tag{15}$$

that is the standard modularity function. This function can be rewritten considering that only the vertex pairs in the same community contribute in the sum:

$$Q = \sum_{c}^{|C|} \left(\frac{l_c}{|E|} - \left(\frac{k_c}{2|E|} \right)^2 \right) \tag{16}$$

where l_c is the sum of edges that connect nodes in c and k_c is the sum of degree of nodes that belongs to c, i.e. total degree.
The modularity function Q it is in range [-1/2, 1] [14], and if we consider the whole graph as a unique community c we obtain $Q = 0$. Opposite, if we consider each nodes as community, $Q < 0$. Then, if a partition has a modularity score < 0, the partition hasn't a modularity structure.

3.3.2 Resolution Limit

There is a well-known limit of the modularity function, identified by Fortunato and Barthélemy [11] in 2006. Considering (14), we can easily compute the expected number of edges P_AB between two clusters c_A and c_B, that are separate cluster in partitions C, as:

$$P_{AB} = k_A k_B / 2m \tag{17}$$

where k_c is the total degree of c. We can compute from (16) the difference ΔQ_{AB} that affecting the modularity when we consider c_A and c_B in a partition where they are two different cluster respect to the partition where they are merged in one cluster c_{AB}:

$$\Delta Q_{AB} = \frac{l_{AB}}{|E|} - \frac{k_A k_B}{2|E|} \tag{18}$$

where l_{AB} is the sum of edges that connect nodes that belongs to A to nodes that belongs to B. Now considering the case $l_{AB} = 1$: there is only one edge that connects these two clusters. Therefore we expect that we obtain a greater modularity score keeping these two clusters separate respect to merging them. Instead, from (18) we have that the modularity increase if $\frac{k_A k_B}{2|E|} < 1$. For the sake of simplicity, we assume that $k_A = k_B = k$. We obtain that if $k < \sqrt{2|E|}$, the modularity is greater if we merge the communities. From this it follows that if the communities are sufficiently small in degree, the expected number is smaller than one: in this case if there is only one edge between the two communities, we obtain a better result merging them. The result of this observation is that the modularity optimization has a resolution limit that prevents it to detect communities that are smaller respect the graph as a whole. This problem has many implications: the real networks graph have a community structure composed by communities very different in size, so some of this community may be wrongly merged. Fortunato identifies as week point the assumption that in the null model each vertex can interact with every other vertex [15]. Some solutions are proposed, as tunable parameters that allow avoiding the problem or also algorithm that eliminate artificial mergers. By the way, in many real cases, the modularity-based algorithms still obtain a very good result and permit to analyze quickly very large graph. For those reasons, the algorithms of this class of algorithm remain the most used, but it's important to remark their limits.

3.4 Girvan and Newman algorithm

Now we present the Girvan and Newman algorithm [6]. This method deserves to be presented because it is the first method that uses the modularity as quality function [8] and in some sense represents a turning point in the history of community detection. This method is a divisive algorithm, i.e. it tries to identify edges that connect two communities and then remove that edge. The goal of the algorithm is to get clusters disconnected from each other. To select which edge we have to remove, we introduce the concept of edge betweenness. The edge betweenness it is a measure that quantifies how an edge is least central for a community. If an edge connected two communities, it should have a greater value compared to an edge that is incident to two nodes that are in the same community.
The algorithm has 2 steps iterated until all edges are removed:

1. computation of the edge betweenness for each edge;

2. removal of the edge with the largest betweenness;

The algorithm construct an entire dendrograms of partitions, and the modularity is used to select the best one.

Girvan and Newman proposed three different definitions of edges betweenness [8]: shortest-path, current-flow and random walk. The first one is the number of shortest paths between all vertices which contributes the edge. The computation of this value for each edge of the graph has a complexity $O(n^2)$ on a sparse graph [8]. The second definitions consider the graph as a resistor network created by placing a unit resistance on every edge of the network. If a voltage difference is applied between any two vertices, each edge carries some amount of current. The current flows in the network are governed by Kirchhoff's equations and the calculations are performed on each edge in the graph. This calculation has a complexity $O(n^3)$ on a sparse graph [8]. The last one is the expected frequency of the passage of a random walker on the edges. The calculation requires the inversion of the adjacency matrix followed by the calculus of the averaging flows for all pairs of nodes. The complexity is $O(n^3)$ on a sparse graph [8]. The first definition is the most used for its speed ($O(n^2) < O(n^3)$) and it is also shown that in practical application this edge betweenness gives better results [8]. The authors also show that the recalculation step is essential to detect correctly communities: this means that we have to recalculate the betweenness every time an edge will be removed, raising the complexity of the algorithm to $O(n^3)$ on a sparse graph. The complexity is the strongest limit of this algorithm, which, however, was the first one to introduce the modularity and has many ideas that were used later on.

3.5 Modularity Optimization Techniques

After the introduction of the modularity function Q, many algorithm were presented in the literature that have as goal to optimize the modularity function Q. In this chapter we present the Newman's greedy algorithm that was the first one, in order to make a comparison with the Louvain algorithm that is also greedy. This algorithm also introduce for the first time the concept of ΔQ, that will be expanded and used in the Louvain method. Moreover, we present also some other class of techniques that are used in modularity optimization like extremal optimization, simulated annealing and spectral clustering.

3.5.1 Greedy Method of Newman

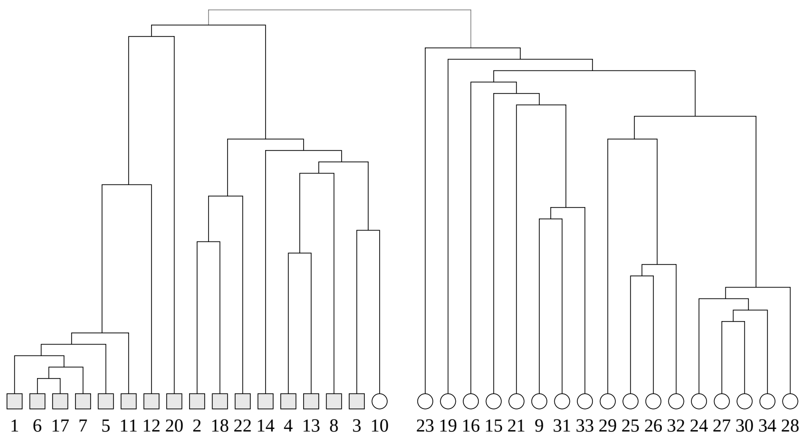


Figure 10: Dendrogram of the communities found by Newman algorithm in Zachary karate club network. This image is reprinted from [8].

The first modularity optimization algorithm is presented by Newman [8]. and it is an agglomerative hierarchies methods. Given a graph $G(V, E)$ where n are the number of nodes and m the number of edges, the algorithm starts with all nodes assigned to each own cluster and no edges between them. The first step of the algorithm is to pick from the set of vertices the vertices that give the maximum increase (or minimal decrease) of the modularity respect to the actual configuration. This value is indicated as $\Delta Q = Q_{now} - Q_{old}$ The modularity will be calculated on the full graph and not on the "cluster" graph. Then we add the edge to the graph. If the edges connect two sets of unconnected edges, it delivers a new partition and reducing by one the number of the partitions. So, the algorithm find n different partitions of the graph (Figure 10). We make some consideration of this procedure:

- If we add some edges that don't merge any partitions (i.e. it is internal), the modularity doesn't change.
- Considering this, we have to calculate the modularity difference ΔQ only when we merge different partitions and so this operation is executed n times.
- Computing Q requires a time of $O(m)$ that became $O(n)$ on a sparse graph.

For those reasons, the complexity of this algorithm is $O(n^2)$ on a sparse graph. Many improvements of this algorithm were proposed later (like the Clauset et. al versions [7] that using a max-heap to reduce the complexity to $O(n \log_2(n)))$ but the complexity of the algorithm remains the biggest limit of it, even if this algorithm still allows to analyze large graphs.

3.6 Other techniques

The previous and Louvain algorithms are the two most famous greedy algorithm of community optimization, but other optimization strategies were proposed in the literature. A class of techniques are based on the concept of simulated annealing, i.e. an exploration of the space of the possible configuration looking for the maximum of Q. Transitions between states are performed combining two types of "move": the first one assign a vertex to a cluster chosen randomly; the second one merge or split communities [10]. These methods reach a very high score of modularity, near to the maximum. In opposite, it's very slow [15].
To overcome this time problem, a heuristic denominated extremal optimization (EO) was proposed to perform an exploration of the space quickly. We define as fitness function F of the vertex x is the local modularity of x divide by its degree. Starting from a random equal size bi-partitions of nodes, at each iteration a node was picked with a probability proportional to the score of the fitness measure is assigned to the other cluster. When there is no more improvement in modularity, the algorithm was called recursively on the two clusters. With a total complexity of $O(n^2 \log(n))$, this algorithm is a good trade-off between accuracy and speed [9].
Finally, in literature one presented the idea of combining modularity optimization with the spectral clustering. Given the adjacency matrix A of the graph G, we define the matrix B whose elements are:

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2|V|} \tag{19}$$

Modularity can be optimized by using spectral bisection on the matrix B [15]. This algorithm has a total complexity of $O(n^2 \log(n))$.

4 Louvain Algorithm

The Louvain algorithm is a greedy modularity optimization techniques created from a team of researcher from Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte and Etienne Lefebvre in the 2008 [13]. The algorithm bears the name of the university to which they belong to, i.e. *Université Catholique de Louvain.* In 2008, the fastest algorithm presented in the literature was the one proposed by Clauset et al. [7], but the biggest graph at the time that was analysed has 5.5 million users. This was a not so big graph even at the time. For example, Facebook in 2008 has 64 million active users, more than ten times the size of the biggest analyzed graph. This algorithm was proposed to resolve this scaling problem: indeed the first version of this algorithm identifying communities in a 118 million nodes network in 152 minutes [13]. From that year, many improvement was made and some parallel versions were proposed. This algorithm and its parallel version is the main topic of this thesis. The algorithm is very popular due to his simplicity, efficiency and overall precision. In this chapter, we present the sequential algorithm in details and some optimization technique presented in the literature. Then we present the parallel version of the algorithm, focusing on the implementations dedicated to the GPU.

4.1 Description

This greedy algorithm its quite simple. There are two phases that are repeated iteratively: the optimization phase and the aggregation phase. At the start of the optimization, each nodes is assigned to its self-community, i.e. each node belongs to a community composed by only itself. In the first phase, for each node i, we evaluate for each community j that have at least one node in the neighbour of i $N(i)$, the gain of modularity $\Delta Q_{i \to c_j}$ that we have if we remove i from its community c_i and we assign it to C_j. To due this, we can use the equations (16) to calculate the modularity in current configuration $Q_{i \to c_i}$ and the modularity $Q_{i \to c_j}$ in the configuration where i is assigned to c_j and subtract, but this is quite inefficient. Instead, we can calculate

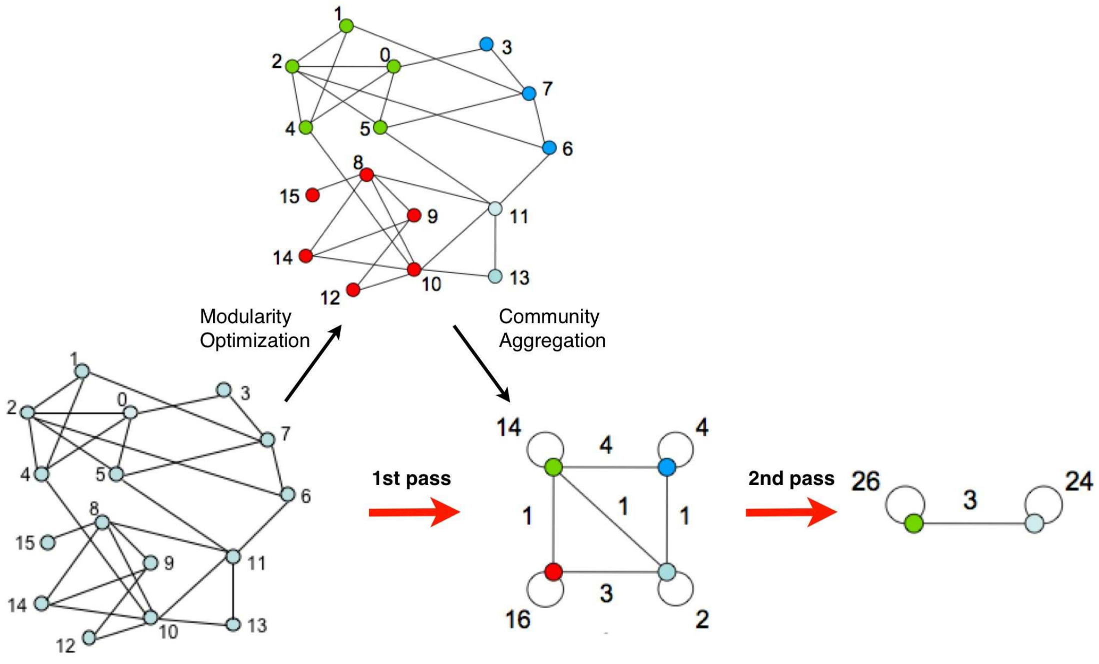


Figure 11: Scheme of the Louvain algorithm. This image is reprinted from [13].

directly $\Delta Q_{i \to c_j}$:

$$\Delta Q_{i \to c_j} = \frac{l_{i \to c_j} - l_{i \to c_i/\{i\}}}{2|V|} + k_i \frac{k_{c_i/\{i\}} - k_{c_j}}{4|V|^2} \tag{20}$$

where $l_{i \to c_j}$ is the sum of edges that connect i to the community c_j, k_i is the weight of the nodes i and k_{c_j} is the weight of the community c_j. Then we define the subset Z_i the set of community c_z with $z \in N(i)$ such that:

$$\Delta Q_{i \to c_z} \geq \Delta Q_{i \to c_j} \qquad \forall j \in N(i) \tag{21}$$

If more there is more than one community in the group, one community c_z^* was selected using a braking rule, otherwise we pick the only community in Z_i. If $\Delta Q_{i \to c_z^*} > 0$, we move the node i to the community c_z^*.
This process is applied repeatedly and sequentially for all nodes while modularity score increases. When no more improvement can be reach, the second phase start. In this phase a new network was created from the results of the previous phase: in the new graph, the nodes are the communities found, and the edge between them are given by the sum of the links between nodes that belong to the corresponding

communities (edge between nodes in the same communities lead to self-loop). Then we reapply the first step and then the second one until no more improvement is obtained. An example of the algorithm is shown in the Figure 11.
The complexity of this algorithm is $O(m)$ where m is the number of the edges of the graph, due to the fact that we can compute the gains in modularity for each neighbour easily. Respect to the previous approach, this techniques reaches the goal of the execution in linear times. Indeed, this algorithm can create an entire hierarchies of partitions and this can be useful to avoid the resolution limit problem: we can analyze in the dendrogram the intermediate solutions to observe its structure with the desired resolution [13].

4.2 Pruning

This algorithm even in the first formulation is quite efficient, but large network analysis requires improvement to be executed quickly. The parallel techniques are very useful for this task and it will be presented in the next chapter. Now we focus on a method that speed-up the computations in the sequential field but that is also suitable in parallel.
The first optimization phase is the most time consuming ones [13], consuming about 80% of the time [17]. To reduce the impact of this first phase, in literature were proposed various approach. For example, in [21], V. A. Traag proposed to randomize the choice of the community which you want to assign the nodes between the communities of the neighbour. The idea behind these techniques is that the nodes tend to be in a "good". This technique performs well sequentially if the graph were the community structure is well defined. Instead, in parallel behaviour, this method doesn't be so good due to the fact, each node changes communities simultaneously, and there is no way to prevent simultaneous swaps without introducing some overhead and this may lead to a convergence problem. For this reason, we choose another technique more parallel friendly, introduced by Ozaki et. al [23]. Now we present this simple and efficient technique of optimization for this algorithm that doesn't afflict the quality of the partitions. This method makes a pruning of the nodes in the optimization phase in order to compute the maximum delta modularity for

only the nodes that have the potential to change community. Every time a node i changes community from X to Y, its affect the ΔQ of its neighbourhood and all nodes linked and in X and Y. Referring to 16, we describe all these cases:

- Nodes in X that aren't connected to i: for those nodes, the value of ΔQ_X increase because of the degree of the community k_X decrease without affecting the value of l_X.
- Nodes in Y that aren't connected to i: for those nodes, the value of ΔQ_Y decrease because the degree of the community k_Y increase without affecting the value of l_Y.
- Nodes that are linked to a node in X, but not to i: for those nodes, the value of ΔQ_X increase because of the degree of the community k_X decrease without affecting the value of l_X.
- Nodes that are linked to a node in Y, but not to i: for those nodes, the value of ΔQ_Y decrease because the degree of the community k_Y increase without affecting the value of l_Y.
- Nodes that are linked to i in X: in this case both k_X and l_X decrease for ΔQ_X..
- Nodes that are linked to i in Y: in this case both k_Y and l_Y increase for ΔQ_Y.
- Nodes that are linked to i but that are not either in X or in Y: in that case afflict both ΔQ_X and ΔQ_Y (increase k_X, l_X, k_Y and l_Y).

The nodes considered in first and the fourth case, doesn't have the potential to change community: in the first case one increase the value of ΔQ_X that is the maximum (because they are already in the community X); in the fourth case one decrease the values of ΔQ_Y that aren't the maximum (because they are not in the community Y). In all other cases, there is a chance that some nodes change community. In the Figure 12, the white nodes are the nodes that doesn't have the potential to change community, instead the black and striped ones are the ones that may have. Considering only this nodes, the computation time will be reduced

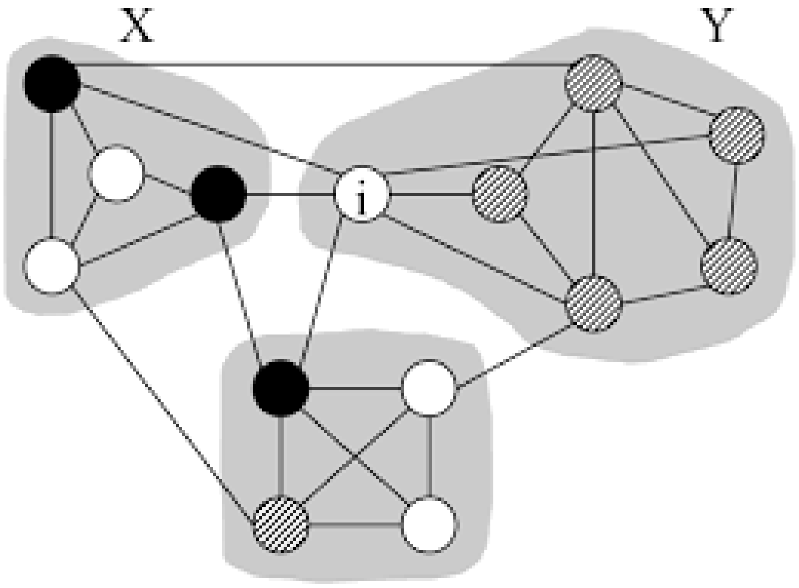


Figure 12: Example of graph where the nodes i changed community from X to Y. This image is reprinted from [23].

without reducing the quality of the partition.
The optimization proposed by Ozaki et. al consists to create a set of nodes during the iteration of the optimization that will be analyzed in the next step: at the start of the optimization phase, an empty set S is created and every time a node i change its community, each nodes in its neighbourhood that doesn't belong to the new community is add to S. The next iteration consider only the nodes in S and the process is iterated. They consider only one of the four previous categories of nodes: this is because calculating all nodes (explicitly the ones in the second and third group) introduce overhead and this group is the most influential for ΔQ [23]. The selected nodes to be add to S are the black one in the Figure 12. The experimental result show that reduce the computational time by up to 90% compared with the standard Louvain algorithm. In terms of accuracy, surprisingly, the modularity is almost the same, not only the final one, but also the transition of the modularity during the iterations [23].

4.3 Parallel Implementations

Now we present various approach that was used in literature to improve the performance of the Louvain algorithm. We can divide the parallelization techniques in two different class: the coarse grained approach and the fine grained approach. The methods in the first class divides the nodes in some sets and the modularity are processed sequentially independently for each set. When all sets are analyzed, the algorithm merge the results for the next phase. Instead, the second approach consider each nodes independently. The best modularity were calculated for each node simultaneously, therefore the decision of the new community for each node is based on the previous configuration. Wickramaarachchi et al. [17] proposed one of the first coarse algorithm: in the first iteration, the algorithm partition the graph in subgraphs and the execution were performed simultaneously and independently on each partition. Edges that cross the partition were ignored. In terms of quality, they showed that ignoring edges cross partition edges does not impact to the quality of the final result.

In 2015, both Staudt and Meyerhenke [20] and Lu et al. [18] proposed an fine grained implementation based on OpenMP. To compute $Q_{i \to c_j}$ for each nodes i and each communities c_j in neighbourhood of i , the algorithm must calculate $l_{i \to c_j}$(i.e. the sum of edges that connect i to the community c_j). These values may change in every new configuration: for this reason we must have a method to get it them fast. In [20], they try to associate each node with a map in which the edge weight to neighbouring communities was stored and updated when node moves occurred, but they discover that introduced too much overhead. Instead, recalculating each time the weight to neighbour communities each time a node is evaluated turned out to be faster. Therefore, they proposed to use a `map` for each node as accumulator of his edges to calculate every $l_{i \to c_j}$. Instead the total weights of each community k_{c_j} is stored and updated every time a nodes change community. The same scheme is used in [18]. A more complex schema was proposed by Que et al. [19]: they proposed an algorithm based on a communication pattern that permit to propagate the community state of each nodes. Due to his complex behaviour, this schema is hard to implement on the GPU.

Forster in [22] presented a GPU implementation based on the first two previous OpenMP version: he reports a speed-up to a factor of 12, but in the paper there isn't information about the quality of the partition. Following, the algorithm preposed form Naim et al. [24] parallelize the hashing of the edges both in optimization and also in the aggregation phase. In addition, they partitioning the vertices into subsets on their degrees in order to obtain an even load balance between threads. A different implementation was proposed by Cheong et al. [16]: it is a multi-GPUs implementation that used a coarse grain model between the GPUs and than a fine grain model for the computation of the modularity of each sub-graphs. This algorithm its also peculiar because doesn't use hashing to calculate the modularity but sort each neighbour list based on the community ID of each neighbouring vertex.

5 The GPU's Algorithms

5.1 Fast-Sort-Reduce

5.2 Hashmap Version

6 Performance and Analysis

7 Conclusion

References

[1] D.R. Fulkerson L.R. Ford. “Maximal Flow Through a Network”. In: *Canad. J. Math. 8* (1956).

[2] A.Rényi P.Erdös. “On Random Graphs”. In: *Publ. Math. Debrecen 6* (Dec. 1959), pp. 290–297.

[3] Derek J. de Solla Price. “Networks of Scientific Papers”. In: *Science* 149.3683 (1965), pp. 510–515. ISSN: 0036-8075. DOI: 10.1126/science.149.3683.510. eprint: https://science.sciencemag.org/content/149/3683/510.full.pdf. URL: https://science.sciencemag.org/content/149/3683/510.

[4] W.W. Zachary. “An information flow model for conflict and fission in small groups”. In: *Journal of Anthropological Research* 33 (1977), pp. 452–473.

[5] Albert-László Barabási and Réka Albert. “Emergence of Scaling in Random Networks”. In: *Science* 286.5439 (1999), pp. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. eprint: https://science.sciencemag.org/content/286/5439/509.full.pdf. URL: https://science.sciencemag.org/content/286/5439/509.

[6] M. Girvan and M. E. J. Newman. “Community structure in social and biological networks”. In: *Proceedings of the National Academy of Sciences* 99.12 (June 2002), pp. 7821–7826. ISSN: 1091-6490. DOI: 10.1073/pnas.122653799. URL: http://dx.doi.org/10.1073/pnas.122653799.

[7] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. “Finding community structure in very large networks”. In: *Physical Review E* 70.6 (Dec. 2004). ISSN: 1550-2376. DOI: 10.1103/physreve.70.066111. URL: http://dx.doi.org/10.1103/PhysRevE.70.066111.

[8] M. E. J. Newman and M. Girvan. “Finding and evaluating community structure in networks”. In: *Physical Review E* 69.2 (Feb. 2004). ISSN: 1550-2376. DOI: 10.1103/physreve.69.026113. URL: http://dx.doi.org/10.1103/PhysRevE.69.026113.

[9] Jordi Duch and Alex Arenas. "Community detection in complex networks using extremal optimization". In: *Phys. Rev. E* 72 (2 Aug. 2005), p. 027104. DOI: 10.1103/PhysRevE.72.027104. URL: https://link.aps.org/doi/10.1103/PhysRevE.72.027104.

[10] Roger Guimerà and Luís A. Nunes Amaral. "Functional cartography of complex metabolic networks". In: *Nature* 433.7028 (Feb. 2005), pp. 895–900. ISSN: 1476-4687. DOI: 10.1038/nature03288. URL: http://dx.doi.org/10.1038/nature03288.

[11] S. Fortunato and M. Barthelemy. "Resolution limit in community detection". In: *Proceedings of the National Academy of Sciences* 104.1 (Dec. 2006), pp. 36–41. ISSN: 1091-6490. DOI: 10.1073/pnas.0605965104. URL: http://dx.doi.org/10.1073/pnas.0605965104.

[12] Pall F. Jonsson et al. "Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis". In: *BMC Bioinformatics* 7.1 (Jan. 2006), p. 2. ISSN: 1471-2105. DOI: 10.1186/1471-2105-7-2. URL: https://doi.org/10.1186/1471-2105-7-2.

[13] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/p10008. URL: http://dx.doi.org/10.1088/1742-5468/2008/10/P10008.

[14] Ulrik Brandes et al. "On Modularity Clustering". In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pp. 172–188. DOI: 10.1109/TKDE.2007.190689.

[15] Santo Fortunato. "Community detection in graphs". In: *Physics Reports* 486.3-5 (Feb. 2010), pp. 75–174. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2009.11.002. URL: http://dx.doi.org/10.1016/j.physrep.2009.11.002.

[16] Chun Yew Cheong et al. "Hierarchical parallel algorithm for modularity-based community detection using GPUs". In: *European Conference on Parallel Processing*. Springer. 2013, pp. 775–787.

[17] Charith Wickramaarachchi et al. “Fast parallel algorithm for unfolding of communities in large graphs”. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2014, pp. 1–6.

[18] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. “Parallel heuristics for scalable community detection”. In: *Parallel Computing* 47 (2015), pp. 19–37.

[19] Xinyu Que et al. “Scalable community detection with the louvain algorithm”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 28–37.

[20] Christian L Staudt and Henning Meyerhenke. “Engineering parallel algorithms for community detection in massive networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (2015), pp. 171–184.

[21] V. A. Traag. “Faster unfolding of communities: Speeding up the Louvain algorithm”. In: *Phys. Rev. E* 92 (3 Sept. 2015), p. 032801. DOI: `10.1103/PhysRevE.92.032801`. URL: `https://link.aps.org/doi/10.1103/PhysRevE.92.032801`.

[22] Richard Forster. “Louvain community detection with parallel heuristics on GPUs”. In: *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*. IEEE. 2016, pp. 227–232.

[23] Naoto Ozaki, Hiroshi Tezuka, and Mary Inaba. “A simple acceleration method for the Louvain algorithm”. In: *International Journal of Computer and Electrical Engineering* 8.3 (2016), p. 207.

[24] Md Naim et al. “Community detection on the GPU”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 625–634.

[25] *Cuda Programming Guide*. 2018. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[26] *NVIDIA TURING GPU ARCHITECTURE*. 2018. URL: `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`.

[27] Yifan Sun et al. *Summarizing CPU and GPU Design Trends with Product Data*. 2019. arXiv: `1911.11313 [cs.DC]`.