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 quality. 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 150 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. s

Contents

1 Introduction

The community detection problem is one of the most interesting field of the graph analysis. In several real world scenarios, we have the necessity of cluster some data considering the relations between them: one example can be the problem of dividing social network users in group by the mutual friendship relation in order to propose targeted advertisings. A natural way to represent this kind of structure based on the relation is a graph, and several technique based on this theory was proposed in the literature in order to solve this kind of problem. Actually, this problem is not easy to solve due to the extremely high number of different possible partition of the data that we can perform, even if the dataset is composed by just few elements. One the most famous and used heuristic is the Louvain algorithm proposed by Blondel et al. [14], due to its speed and its overall quality, even if some limits of it are well known in the literature. This greedy algorithm divides the graph in partitions that maximize a quality function that evaluate this partitions, called modularity. This function is based on the idea that a random graph doesn't exhibit a community structure: therefore, we can evaluate how is well defined the community structure comparing the graph with another graph that keep some structural proprieties but is generated at random, called *null-model* [6]. Given a graph $G(V, E)$, its adjacency matrix A, a partition of nodes C and the corresponding function $c(x)$ that assign each node x to its community, we define the modularity function as follow:

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

where k_i is the degree of the node i and δ is an filter function: its yields one if $c(i) = c(j)$, zero otherwise. The techniques proposed before the Louvain algorithm are quite slow due to the fact that, every time we assign a node to a community, we have to recalculate the values with the previous formula, and this calculation takes a lot of time. To speed up the computation, this algorithm proposed to calculate the variation in modularity assigning a node to another community without recalculate Q with the formula 1. The algorithm is composed by two phases: an optimization phase and a aggregation phase. At the beginning each node is assigned

to a community composed only by itself; in the first we pick each node i of the graph and we calculate for each community c_j in its neighbourhood the values $\Delta Q_{i \to c_j}$, that is the changing in modularity of assigning the node i to the community c_j, as following:

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

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, for each node, we select the maximum values and if it is greater than zero, we assign the node i to the community that correspond to the maximum value. We repeat this procedure sequentially for all nodes while modularity score increases. When no more improvement can be achieved, we execute the aggregation phase. In this phases we create a new graph from the current community structure: each node is one of this communities, 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). After this step we reapply the first step. The algorithm ends up when no more improvement is obtained. This algorithm is quite efficient with a complexity of $O(m)$ where m is the number of edges in the graph, but, nevertheless, this algorithm requires a lot of time to find the communities in the bigger graphs. For this reason, some approach were proposed in the literature to speed up the algorithm. One interesting techniques proposed by Ozaki et. al [28], prune the nodes in the optimization phase, considering only the nodes that have a neighbour that has change its community in the previous iteration. Apart of this kind technique, generally the most effective method to improve significantly the performance of an algorithm is to parallelize the execution. Especially, to obtain the maximum speed, many framework that allow to perform on the GPU computation normally handled by the CPU became popular. The most used framework is the Nvidia CUDA, a parallel computing platform and application programming interface model. CUDA was originally designed as a C++ language extension that allow any developer to building application designed for the GPU with a low learning curve. CUDA is also designed to transparently scale on different GPU.

In literature, few implementation of the Louvain algorithm for the GPU were pro-

posed. In this thesis, we present three new algorithm for the GPU based on the Louvain algorithm: PSR-Louvain, PH-Louvain and Adaptive Louvain. All these algorithm implements the pruning techniques proposed by Ozaki. No other Louvain based algorithm for the GPU implements this technique before. These algorithm compute the maximum score ΔQ for each node simultaneously, and then we update the community based on that score. Also the creation of the edges of the new graph in the aggregation phase are executed in parallel. The first two algorithm are quite similar each other. In both algorithm we start from the edge list composed by the tuple (i, j, w) where i is the source node of the edge, j is the destination node of the edge and w is the weight of the edge. Both algorithm, in the optimization phase, first select only the edges such that the node i have a neighbour that has change its community in the previous iteration, than they substitute each destination node with the corresponding community c_j. After them, the two algorithm using different method to sum up all the values w for each pair i and c_j and obtain the corresponding $l_{i \to c_j}$. We need this value to compute all the $\Delta Q_{i \to c_j}$, as shown in the Formula 2. The first algorithm sorts the list and then performs a segmented reduction to each value with the same pair i and c_j; the second algorithm use an hashmap to aggregate all of this values. After than, both select the maximum value for each node and eventually update the community. Finally, they compute if a node has a neighbour that change its community and store the results that will be used in the next iteration for the pruning. These two algorithm use the same aggregation scheme proposed in the optimization phase to create the new graph in the aggregation phase. In both algorithm is also present an technique that optimize the first iteration of each optimization phase.

This two algorithm, compared to the fastest sequential versions, obtain a speed up factor up to 56. During the analysis of this two algorithm, we discover that the PSR-Louvain tends to perform better than the PH-Louvain in the early iteration of the optimization phase; on the other hand, the PH-Louvain approach outperforms the other when the number of different key inserted in the hashmap falls below a given threshold. Besides the PH-Louvain aggregation phase is faster than the PSR-Louvain one.

On this two consideration, we create a new algorithm that combine this two algorithm, and select the best aggregation scheme on the situation. This Adaptive approach performs better than the other, combining the best feature of this two algorithm. Finally we compare this version with the two fastest GPU Louvain based algorithm: the first one is included in the Nvidia library cuGraph; the second one is included in the high performance library for graph analysis Gunrock. Our test highlight that our Adaptive algorithm optimize the memory occupation better than the other two allowing it to compute graph with approximately twice edges. Besides, our algorithm generally perform better than this two algorithm in terms of time.
This thesis is structured as following: in the Chapter 2 of this thesis we present the Nvidia GPUs architecture and the CUDA framework, paying special attention on the concept that are used later on this thesis; in the Chapter 3 we present the field of the community detection and the modularity optimization, highlighting the motivation that leads to the design of the Louvain Algorithm; in the Chapter 4 we present the sequential Louvain algorithm, the pruning techniques and the previous parallel implementations presented in the literature; then, in Chapter 5, we present in details out two first algorithm, that will be analyzed in details in the Chapter 6; finally, in the Chapter 7, we present the Adaptive Louvain Algorithm and we analyze it, comparing it before with our other two algorithms, then with the two algorithm included in the two libraries. In the last chapter, we sum up our work and we highlight some future research areas and development.

2 Nvidia GPUs architecture and CUDA

Scientists, years by years, 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 standard algorithms.

Besides, at the same time, the manufacturers 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 causes serious problems of heat dissipation and data synchronization. To find a solution to this problems, the manufactures start to produce multi-core CPUs: 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.

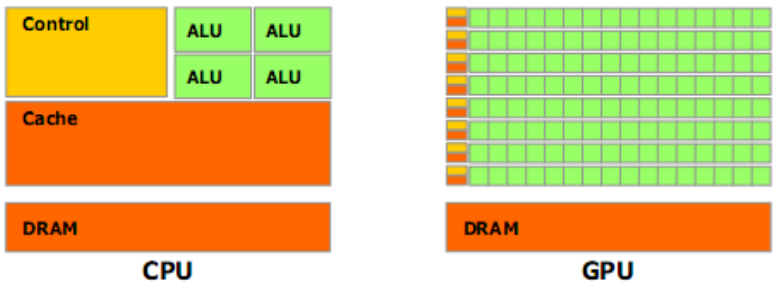


Figure 1: Difference in CPUs and GPUs architecture. This image was reprinted from [30]

The GPU architecture is radically different respect to the CPU. The CPU has sev-

eral 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 [32], from the Figure 2 we can see how the performance of the GPU outclasses the performance on the CPU.

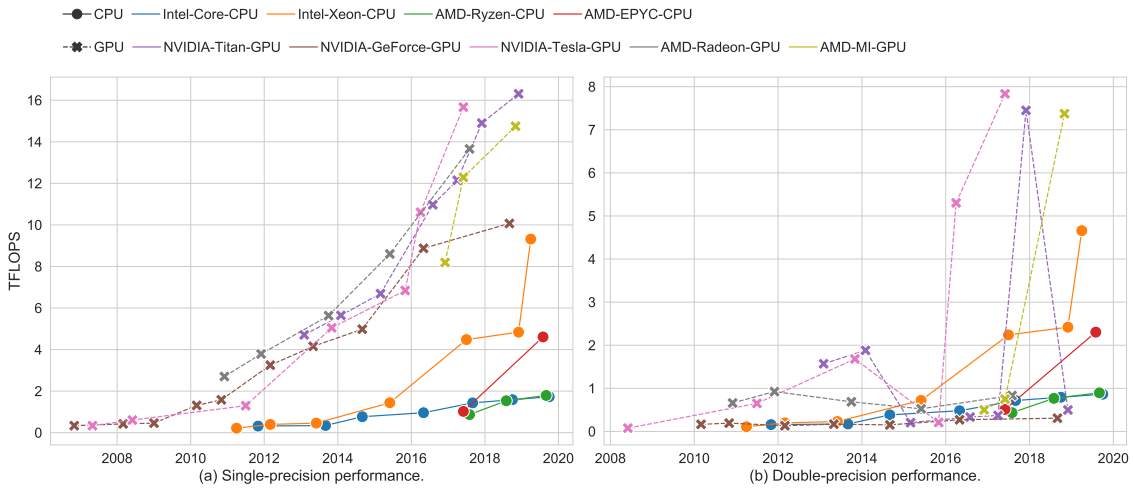


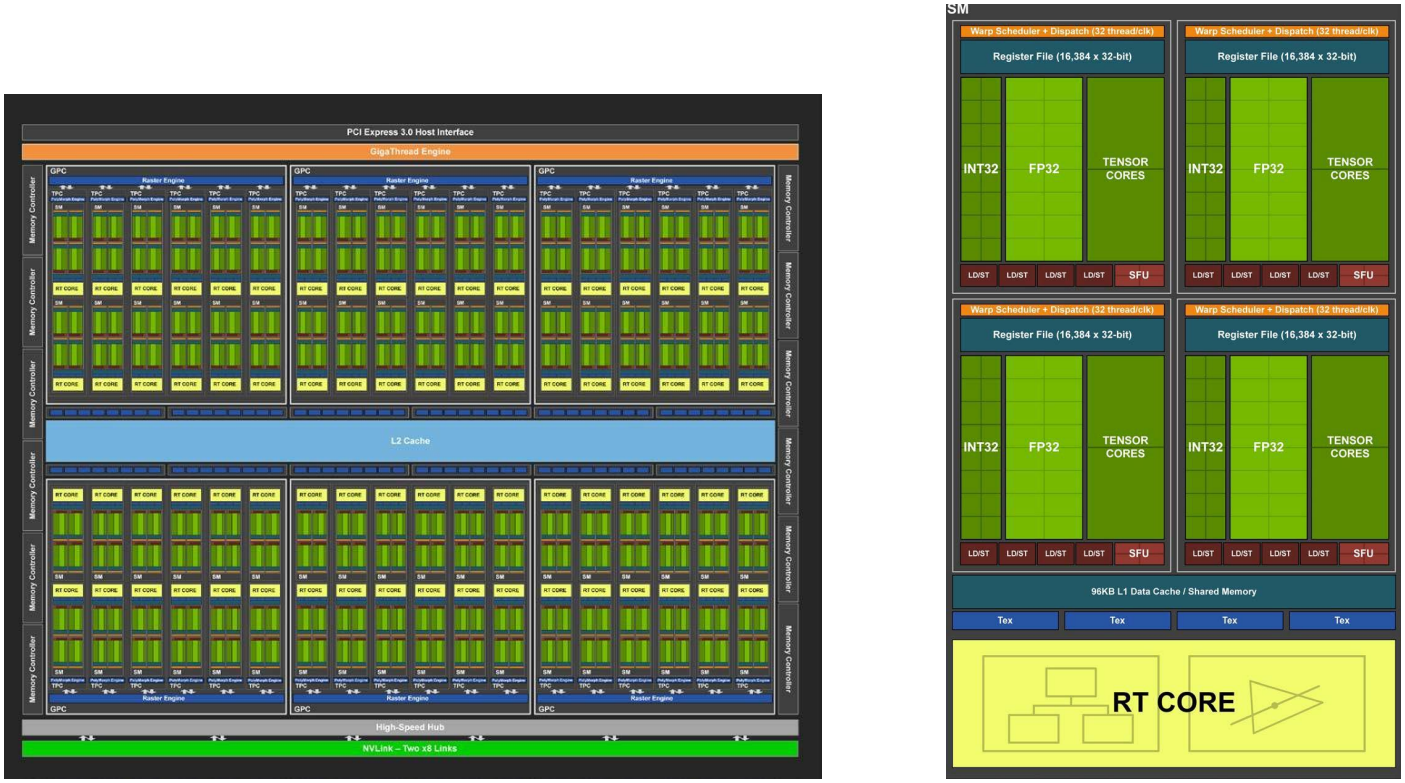
Figure 2: Comparing single-precision and double-precision performance of CPUs and GPUs. The performance are measured in trillion of floating point operations per Second (TFLOPS). This image was reprinted from [32].

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 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, we first present the Nvidia's GPU architecture, then we

present CUDA and Thrust, a parallel computing library. This chapter was based on [31] and [30].

2.1 Nvidia's GPU Architecture

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



(a) (b)

Figure 3: (a): Tuning GPU full architecture; (b) Streaming multiprocessor (SM) in details. Those images was reprinted from [31]

Nvidia's GPU is the concept of Streaming Multiprocessor (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 Turing architecture is present

also a Ray Tracing cores in each SMs: this core is used in rendering.
The SM is the fundamental unit because 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 registers (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 between all components, there is a shared L2 cache, i.e. each thread can have access to it. In the Turing GP, it is large 6144 KB. Therefore, in summary, a Turing TU102 GPU contains 72 SMs and 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 interface aims at providing a framework to the developers that allow building applications that can transparently scale with a low learning curve. To overcome this challenge, CUDA was designed as a C++ language extension: in this way, a programmer that already knows the language syntax can start to develop a GPU-accelerated application with a minimal effort. The support of other language was introduced years by 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

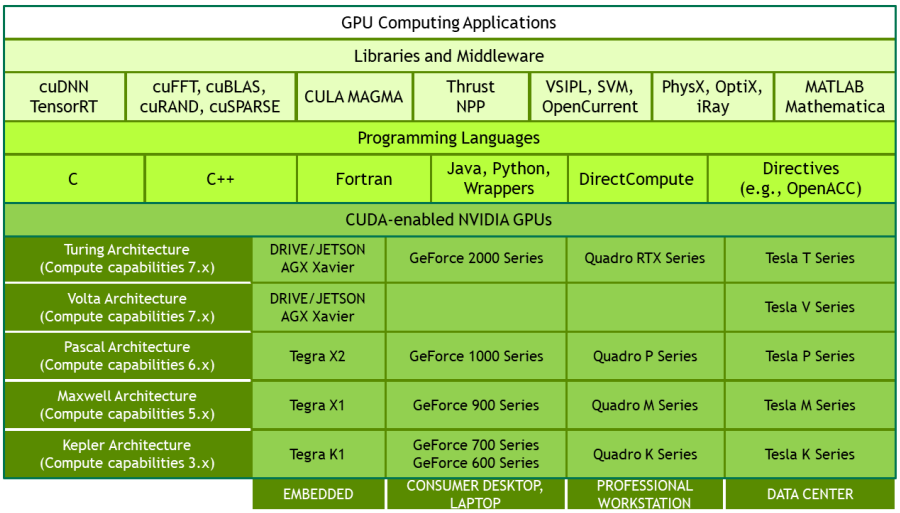
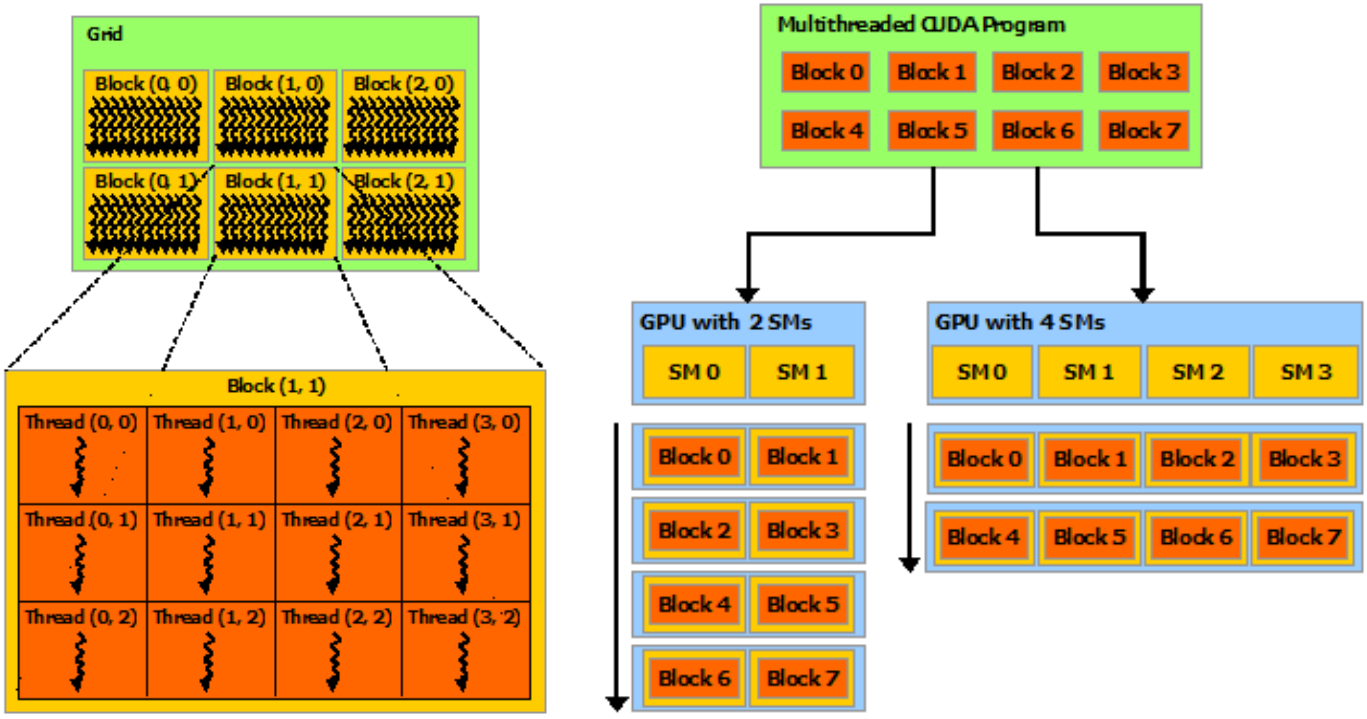


Figure 4: GPU Computing Applications. This image was reprinted from [30].

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 at 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 threads that have to execute the kernel call. Each thread has a unique ID. To set the thread 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 the size 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, each thread have a three-component vector named `threadIdx` that identifies its position in the block. In 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. To define the dimensions of the grids and the dimensions of the blocks in the angle brackets, we use two `dim3` values (or eventually `int` to define a one dimensional 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 the unique ID of the vector from `threadIdx` and `blockIdx`. In Figure 5a is illustrated the grid-blocks schema.



(a) (b)

Figure 5: (a): Grid of Thread Blocks; (b) Automatic Scalability. Those images was reprinted from [30]

As mentioned above, each block is assigned to a different streaming multiprocessor. On current GPUs, a block has a 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 schedules each block 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). On the other hand, 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-block 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 data doesn't permit to parallelize those kernels; moreover, for the sake of completeness, they are well described in [30]). The data must be copied to this memory from

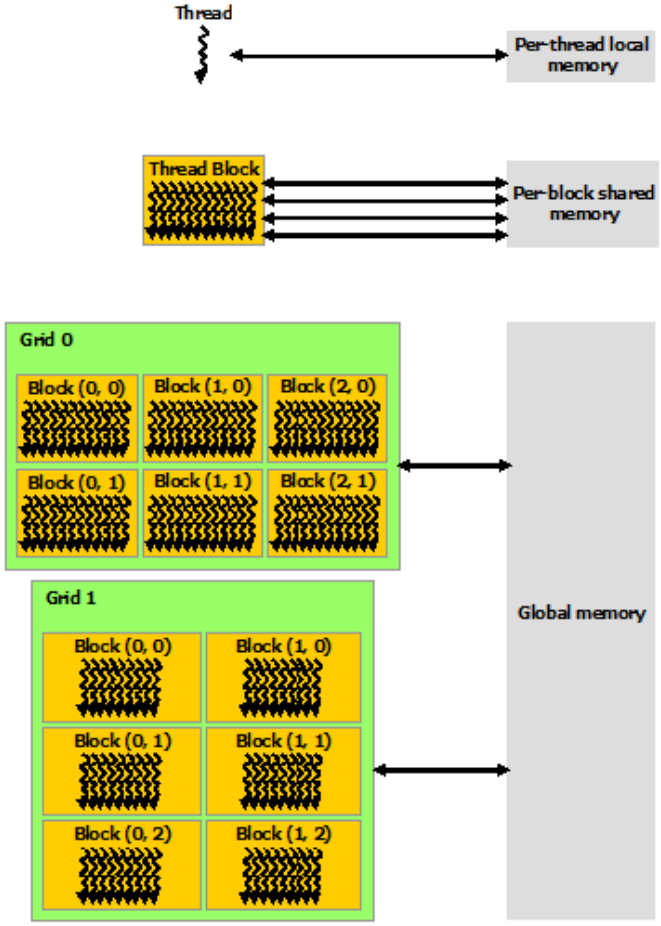


Figure 6: Memory Hierarchy. This image was reprinted from [30].

the host before the kernel execution. Those two types of memory, combined with several primitives that synchronize thread at warp, block or device levels, permit threads collaboration. Those synchronizing function acting as a barrier: all threads in the specific level must wait for the others before any one 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. Our project use a compute capabilities greater than 6.0 .

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 [30].
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 in 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

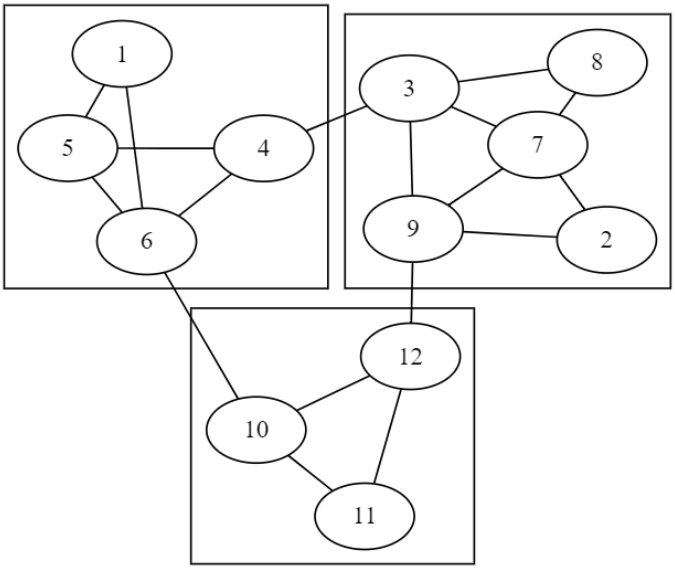


Figure 7: An example of a communities structured graph. Three communities are enclosed by the rectangles.

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 in different research areas: 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.

A study demonstrated that this type of network shown to be useful for highlighting

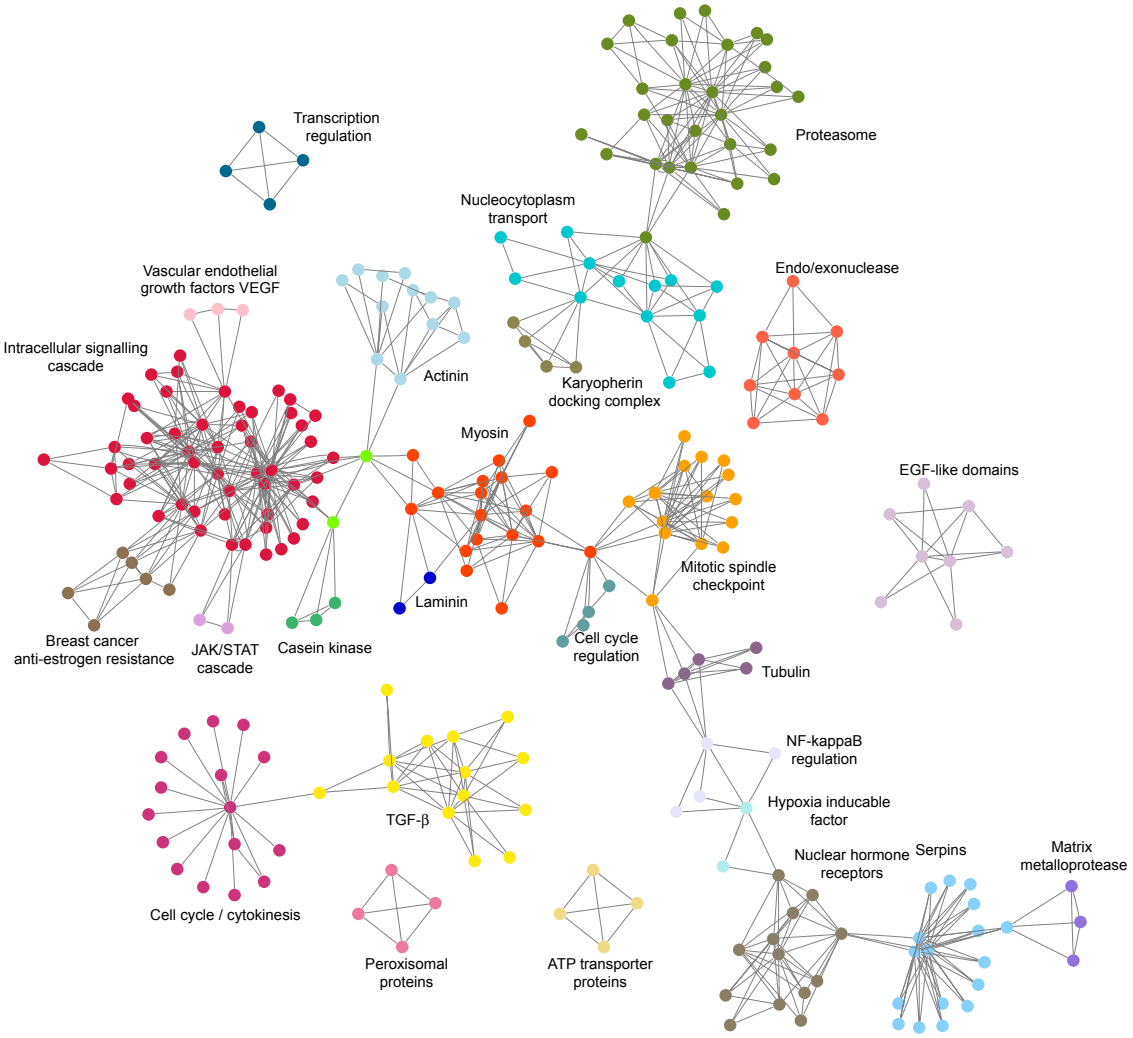


Figure 8: A protein protein iteration network of a rat cancerous cell. This image was reprinted from [13].

key proteins involved in metastasis. [13]

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 used as a reference point to test the detection algorithms used to analyze huge social

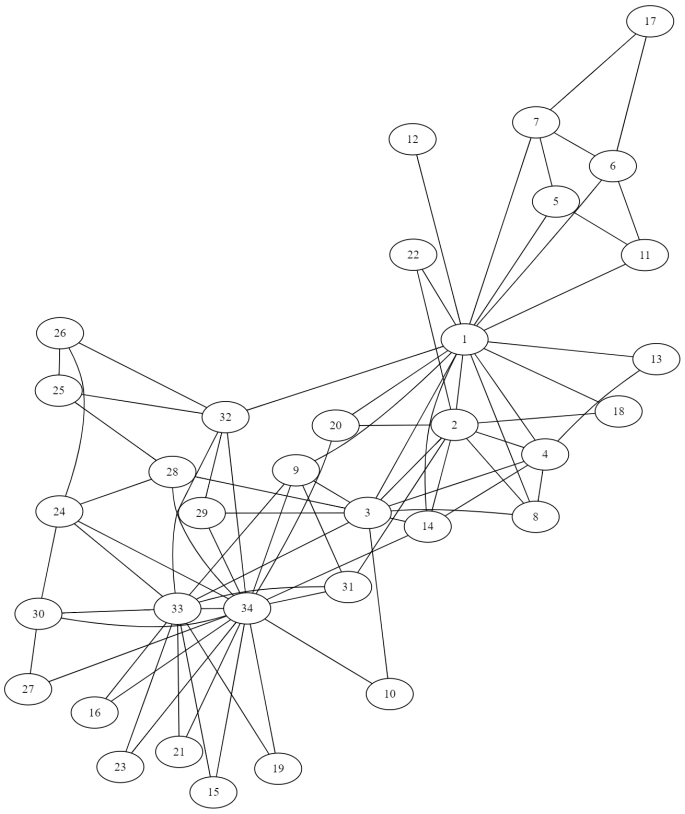


Figure 9: Zacahry's karate club. [4] This image was made with Graphviz.

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 real-world scenarios, all united by the fact that the data is 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 (follow 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 communities 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 [16]. 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 [16] 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. We now present those classes 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 it does not ensures 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 corresponding 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:

$$k_v^{int} = \sum_{k \in C} A_{vk} \qquad\qquad k_v^{ext} = \sum_{k \notin C} A_{vk} \tag{3}$$

where A_{vk} is the entry of A at position (v, k). 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.

$$k_C^{int} = \sum_{i,j \in C} A_{ij} \qquad\qquad k_C^{ext} = \sum_{i \in C, j \notin C} A_{ij} \tag{4}$$

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 quantifies the communities 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 has 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. The comparison between a graph and the corresponding null model, 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 Euclidean space E, we define the norms l_1 (Manhattan distance), l_2 (Euclidian distance) and l_3 (Maximum distance)

as:

$$l_1(a,b) = \sum_{k=1}^{n} |a_k - b_k| \tag{5}$$

$$l_2(a,b) = \sum_{k=1}^{n} \sqrt{(a_k - b_k)^2} \tag{6}$$

$$l_3(a,b) = \max_{k \in [1,2]} |a_k - b_k| \tag{7}$$

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

$$\cos(a,b) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{8}$$

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:

$$d(a,b) = \sqrt{\sum_{k \neq a,b} (A_{ak} - A_{bk})^2} \tag{9}$$

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 [16]. 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 choice 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 [16], 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:

$$\sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - c_i||^2 \tag{10}$$

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 subsets such that it minimizes the edges lying between the clusters. To archive this goal, many algorithms perform 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:

$$\Phi_N(C) = \frac{c(C, V/C)}{k_c} \tag{11}$$

where C is a subset of nodes, k_c is the total degree of C and $c(C, V/C)$ is the sum of all the edges lying between the subsets C and V/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 using the eigenvector derived from the matrix S to cluster the data. 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:

$$L = D - A \tag{12}$$

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. In this situation, there are 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:

$$R = \frac{1}{4} s^T L s \tag{13}$$

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 (14) as follows:

$$R = \sum_{i=0}^{n} a_i^2 \lambda_i \tag{14}$$

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 requires time $O(n^3)$, a too high complexity for big graphs, but there exist some techniques that allow calculating approximate values faster [16].

3.2.4 Hierarchical clustering

The possible partitions of a graph can be very different in scale and some cluster in turn may show an internal community structure. In that case, there is a hierarchy between partitions. The most common way to represent this kind of structure is to draw a dendrogram, i.e. a diagram representing a hierarchical tree. If we draw an horizontal line in the dendrogram, observations that are joined together below the line are in the same cluster (Figure 10). 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 is initially considered as an independent community and the clusters are iteratively merged if the similarity score exceeds a threshold. A divisive algorithm inverts the starting point: at the start, all nodes belong to one

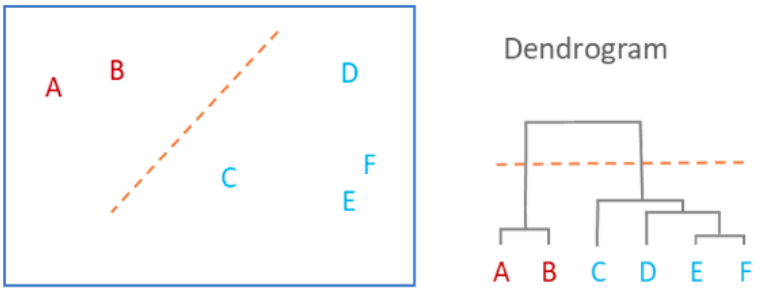


Figure 10: Example of dendrogram. At left we have the data in a Euclidean space, at right we have the dendrogram. The dotted line in the dendrogram divides the data in two cluster, and we show the corresponding line in the Euclidean space.

single community and then the clusters are iteratively split. An example of this type of algorithm, the Girvan and Newman algorithm, is presented later on this thesis. 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: with these algorithm we obtain a entire hierarchies of partitions (from the partition in which each nodes is in a different communities to the one with all the nodes are in a unique community) and we haven't a directed way to isolate the best ones. We need some quality function to find the best partition and the Modularity Function was introduced to overcome this problem. Moreover, as we see in the Girvan and Newman algorithm, building the entire hierarchy using similarity metrics requires 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 [16].

3.3 Modularity Optimization

Historically, the modularity function Q was introduced as a stop criterion for the Girvan and Newman algorithm in 2002. It is a quality function, i.e. a function that allows distinguishing from a "good" clustering 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 defining 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 is 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 their limits in details, the algorithm in which it was firstly used and some optimization techniques based on modularity.

3.3.1 Modularity

The function is based on the idea that a random graph would not exhibit a community structure. We define as *null-model* of a given graph, another graph that is generated randomly yet keeping some structural proprieties of the original one. Comparing the graph with its null model, we can quantify how much the community structure is well defined. 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 node x to its community, we define a generic modularity function as :

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

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 [16]. 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:

$$P_{ij} = 2m p_i p_j = \frac{k_i k_j}{2|E|} \tag{16}$$

Replacing P_{ij} from (16) in (15) we obtain:

$$Q = \frac{1}{2|E|} \sum_{i,j \in V} \left(A_{ij} - \frac{k_i k_j}{2|E|} \right) \delta(c(i), c(j)) \tag{17}$$

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:

$$Q = \sum_{c}^{|C|} \left(\frac{l_c}{|E|} - \left(\frac{k_c}{2|E|} \right)^2 \right) \tag{18}$$

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] [15], 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 [12] in 2006. Considering (16), 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:

$$P_{AB} = k_A k_B / 2m \tag{19}$$

where k_a (k_a) is the total degree of c_a (c_b). We can compute from (18) 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, with respect to the partition where they are merged in one cluster c_{AB}:

$$\Delta Q_{AB} = \frac{l_{AB}}{|E|} - \frac{k_A k_B}{2|E|} \tag{20}$$

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 with respect to merging them. Instead, from (20) 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 too small with respect to the graph as a whole. This problem has many implications: the real networks have a community structure composed by communities very different in size, so some of these communities may be wrongly merged. Fortunato identifies as week point the assumption that in the null model each vertex can interact with every other vertex [16]. Some solutions are proposed, as tunable parameters that allow avoiding the problem or also algorithm that eliminate artificial mergers. Nevertheless, in many real cases, the modularity-based algorithms still obtain very good results and permit to analyze quickly very large graphs. For those reasons, the algorithms of this class 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

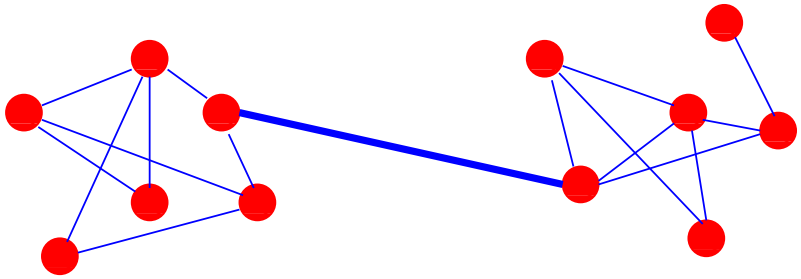


Figure 11: Considering the shortest path definition of the edge betweenness, the highlighted has the much higher values of betweenness than all other edges: indeed all shortest path connecting left vertices and right vertices run through it. For this reason, we chose to remove this edge and we obtain two clusters. This image was reprinted from [**fortunato2007community**].

function [9] and it 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 (Figure 11);

The algorithm constructs an entire dendrogram of partitions, and the modularity is used to select the best one. Girvan and Newman proposed three different definitions of edges betweenness [9]: shortest-path, current-flow and random walk. The first one is the number of shortest paths between all vertices that include the edge (Figure 11). The computation of this value for each edge of the graph has a complexity $O(n^2)$ on a sparse graph [9]. The second definition considers 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 [9]. 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 [9]. 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 [9]. 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 to directly optimize the modularity function Q. In this chapter we present the Newman's greedy algorithm which was the first one, in order to make a comparison with the Louvain algorithm that is also greedy. This algorithm also introduces 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

The first modularity optimization algorithm is presented by Newman [8], and it is an agglomerative method. Given a graph $G(V, E)$ where n are the number of nodes and m the number of edges, the algorithm starts creating a supporting graph that represent the community structure. In this graph at the beginning there are all nodes and no edges between them: this represent the situation in which each node is assigned to a single cluster. The first step of the algorithm is to pick an edge from the original graph to add to the support graph such that it give the maximum increase (or minimal decrease) of the modularity with respect to the actual configuration.

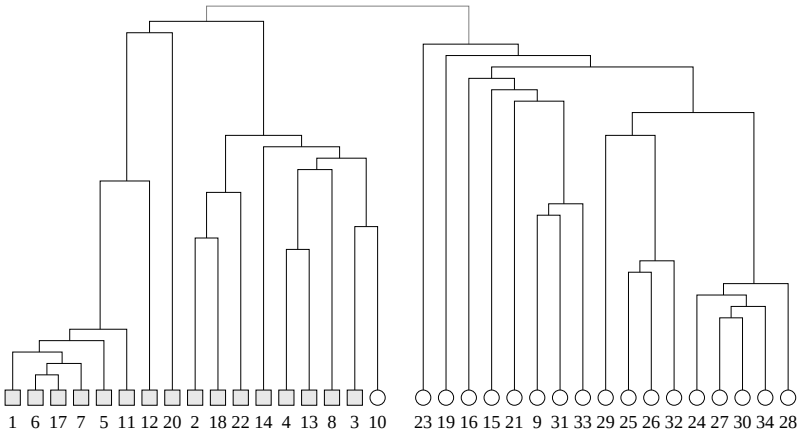


Figure 12: Dendrogram of the communities found by Newman algorithm in Zachary karate club network. This image is reprinted from [8].

This value is indicated as $\Delta Q = Q_{now} - Q_{old}$ The modularity will be calculated on the full graph and not only on the "cluster" graph. Then we add the edge to the support 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 12). 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 version [7] that uses 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 algorithms 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 Q. Transitions between states are performed combining two types of "move": the first one assigns a vertex to a cluster chosen randomly; the second one merges or splits communities [11]. These methods reach a very high score of modularity, near to the maximum. Unfortunately, it's very slow [16].

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 divided by its degree. Starting from a random equal size bi-partitions of nodes, at each iteration a node is picked with a probability proportional to the score of the fitness measure and it is assigned to the other cluster. When there is no more improvement in modularity, the algorithm is 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 [10].

Finally, in literature it was 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:

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2|V|} \tag{21}$$

Modularity can be optimized by using spectral bisection on the matrix B [16]. This algorithm has a total complexity of $O(n^2 \log(n))$.

4 Louvain Algorithm

The Louvain algorithm is a greedy modularity optimization technique designed by a team of researcher composed by Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte and Etienne Lefebvre in the 2008 [14]. 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 identified communities in a 118 million nodes network in 152 minutes [14]. From that year, many improvements were 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 exploiting GPU.

4.1 Algorithm

This greedy algorithm is quite simple. There are two phases that are repeated iteratively: the optimization phase and the aggregation phase. At the start of the optimization, given a graph $G(V, E)$, each nodes is assigned to its self-community, i.e. , each node belongs to a community composed by only itself. In the first phase, given a node $i \in V$, its community c_i and its neighbourhood $N(i)$, we evaluate, for each community c_j such that c_j has at least one node in $N(i)$, 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 the community c_j. We can use the equations (18) 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 compare the difference, but this is quite inefficient. Instead,

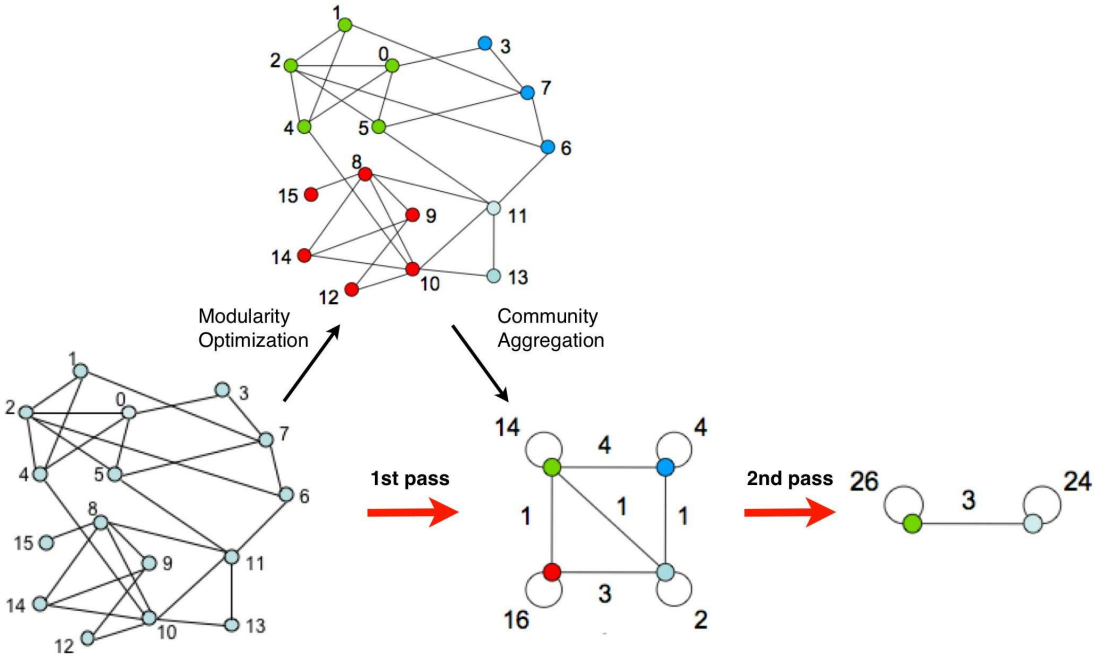


Figure 13: Scheme of the Louvain algorithm. Given a graph, we first execute the modularity optimization phase that assign each nodes assigned to a community. We have 4 different communities, identified by the colours. Then we perform the communities aggregation phase to create a new graph based on the communities found. After that, the first pass is finished and we repeat these steps until we have an improvement in modularity.This image is reprinted from [14].

we can calculate directly $\Delta Q_{i\to c_j}$ as:

$$\Delta Q_{i\to c_j} = \frac{l_{i\to c_j} - l_{i\to c_i/\{i\}}}{2|V|} + k_i \frac{k_{c_i/\{i\}} - k_{c_j}}{4|V|^2} \tag{22}$$

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:

$$\Delta Q_{i\to c_z} \geq \Delta Q_{i\to c_j} \qquad \forall j \in N(i) \tag{23}$$

If there is more than one community in the group, one community c_z^* is 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 sequentially on all nodes, and it is repeated while modular-

ity score increases. When no more improvement can be achieved, the second phase starts. 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 13. 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 time. Indeed, this algorithm can create an entire hierarchy 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 [14].

4.2 Pruning

This algorithm is quite efficient even in the first formulation, but large networks requires improvements 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 speeds up the computation in the sequential field but that is also suitable in parallel.

The first optimization phase is the most time consuming one [14], requiring about 80% of the time [22]. To reduce the impact of this first phase, in literature were proposed various approach. For example, in [26], V. A. Traag proposed to randomize the choice of the community to which assign the nodes. The idea behind these technique is that the nodes that are close to each other tend to be in the same community, so the randomization tend to assign a node to a "good" community. This technique performs well sequentially if the graph has a community structure well defined. Instead, in parallel behaviour, this method doesn't perform well due to the fact that nodes change communities simultaneously: this may lead to a convergence problem and there is no way to prevent simultaneous swaps without introducing

some overhead. For this reason, we choose another technique more parallel friendly, introduced by Ozaki et al. [28]. 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, it affects the ΔQ of its neighbourhood and all nodes linked and in X and Y. Referring to (22), 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

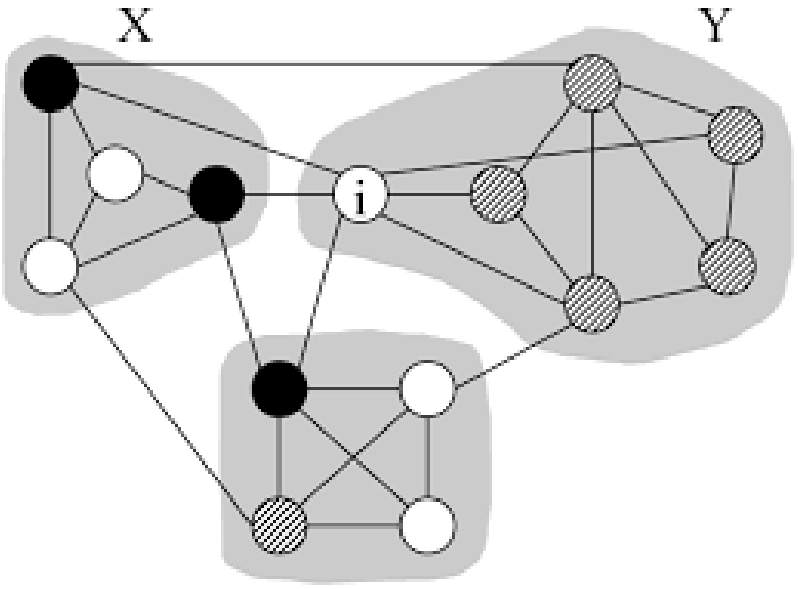


Figure 14: Example of graph where the nodes i changed community from X to Y. The black and striped nodes are the ones with the potential of change community; the white one doesn't have this potential. The pruning technique consider only the black nodes, because considering also the striped ones introducing overhead. This image is reprinted from [28].

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 14, 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 these nodes, the computation time will be reduced without reducing the quality of the partition.

The optimization proposed by Ozaki et. al consists in creating 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 changes its community, all nodes in its neighbourhood that doesn't belong to the new community are added to S. The next iteration considers 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 [28]. The selected nodes to be add to S are the black one in the Figure 14. 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 [28].

4.3 Parallel Implementations

Now we present various approaches that were used in literature to improve the performance of the Louvain algorithm. We can divide the parallelization techniques in two different classes: the coarse-grained approach and the fine-grained approach. The methods in the first-class divide the nodes in some sets and they are processed independently in parallel for each set. The optimization phase of nodes in the same set are executed sequentially. When all sets are analyzed, the algorithm merges the results for the next phase. Instead, the second approach considers each node independently. The community that gives the best modularity for each node is calculated simultaneously and, when all of this values are calculated, we update the communities still in parallel: therefore the decision of the new community for each node is based on the previous global configuration. Wickramaarachchi et al. [22] proposed one of the first coarse-grained algorithms: in the first iteration, the algorithm partition the graph in subgraphs and the execution is performed simultaneously and independently on each partition. Edges that cross the partition are ignored. In terms of quality, they showed that ignoring cross partition edges does not impact the quality of the final result.

In 2015, both Staudt and Meyerhenke [25] and Lu et al. [23] proposed an fine-grained implementation based on OpenMP. To compute $Q_{i \to c_j}$ for each node i and each community 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 them fast. In [25], 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 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 an accumulator of his

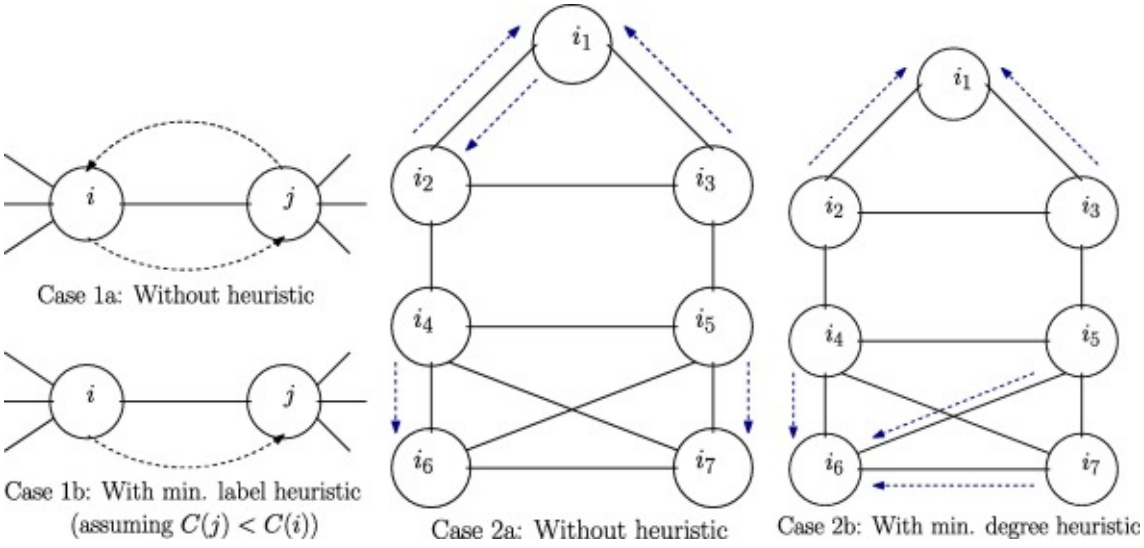


Figure 15: Examples of cases which can be handled by using the minimum labelling heuristic. This image is reprinted from [23].

edges to calculate every $l_{i \to c_j}$. In contrast, the total weights of each community k_{c_j} is stored and updated every time nodes change community. This algorithm obtains a speed-up to a factor of 9 with the same quality of the sequential algorithm. The same scheme is used in [23] and they obtain a speed-up to a factor of 16. This algorithm also highlights a problem of the fine-grained approach: as we can see in the Figure 15 at the case 1a, neighbouring singleton vertices can simultaneously moving to each others communities if they have the same modularity score. To avoid this problem, they define a rules that say: if a vertex i which is in a community by itself c_i decides to move to another community c_j which also contains only one vertex j, then that move will be performed only if $c_j < c_i$. A similar heuristic was used as breaking rules to determine the new community for the nodes: as we see in the in the Figure 15 at the case 2b, it will improve convergence and permit to avoid local maxima.

A more complex schema was proposed by Que et al. [24]: they proposed an algorithm based on a communication pattern that permits to propagate the community state of each node. Due to his complex behaviour, this schema is hard to implement on the GPU.

Forster in [27] presented a GPU implementation based on the first two previous OpenMP version: he reports a speed-up to a factor of 12 respect to the OpenMP

version, but, in the paper, there isn't information about the quality of the partition. Following, the algorithm proposed form Naim et al. [29] 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 to obtain an even load balance between threads. A different implementation was proposed by Cheong et al. [19]: it is a multi-GPUs implementation that used a coarse grain model between the GPUs and then 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: it creates a neighbour community list for each node, sort every list and then sum up the value. In the test with 4 GPUs, this multiple GPUs version is about 3.5 times faster respect to the single GPU version but also reports a loss of up to 9% in modularity.

5 PSR-Louvain and PH-Louvain

In this chapter, we present two novel parallel implementations of the Louvain algorithm: both versions implement the pruning presented by Ozaki et. al. [28]. The two algorithm differ on the way in which they accumulate the edges to calculate $l_{i \to C_j}$ (see formula 18). The first one, the PSR-Louvain, where PSR stands for Prune, Sort and Reduce Louvain, is based on the sort-reduce pattern: it sorts the list and performs a reduction of consecutive values with the same key. We also use a reduction on a sorted array to compute the maximum values of modularity for each node. The second algorithm, PH-Louvain, where PH stands for Prune and Hashmap Louvain, uses a map to accumulate that values. In this chapter, we present firstly the the algorithms, then a special speed-up technique of the first iteration of the optimization phase included in both algorithms and finally the data structure and the implementations details.

5.1 PSR-Louvain

The Prune, Sort and Reduce Louvain algorithm is the first version of the algorithm that we present in this thesis. As the sequential Louvain algorithm, we can divide it into two steps iterated alternately: the optimization phase and the aggregation phase. Furthermore, we divide the optimization phase in eight sub-phases, in which the operations are executed in parallel. This algorithm takes in input a graph stored as a list of edges, where each entry is a tuple (i, j, w): i is the source node, j is the destination node and w is the weight of the edge; in this algorithm, even if the graph is undirected, we consider every edge twice reverting the order of the source and the destination. This list is sorted by (i, j). In the beginning, we have each node is assigned to a community composed only by itself. The weight of each node, a map that associate at each nodes the corresponding communities and the total weight of each communities. At the first iteration each node is in a community by itself and the total weight of each community corresponding at the weight of the unique node in it. The sub-phases of the optimization phase are the following:

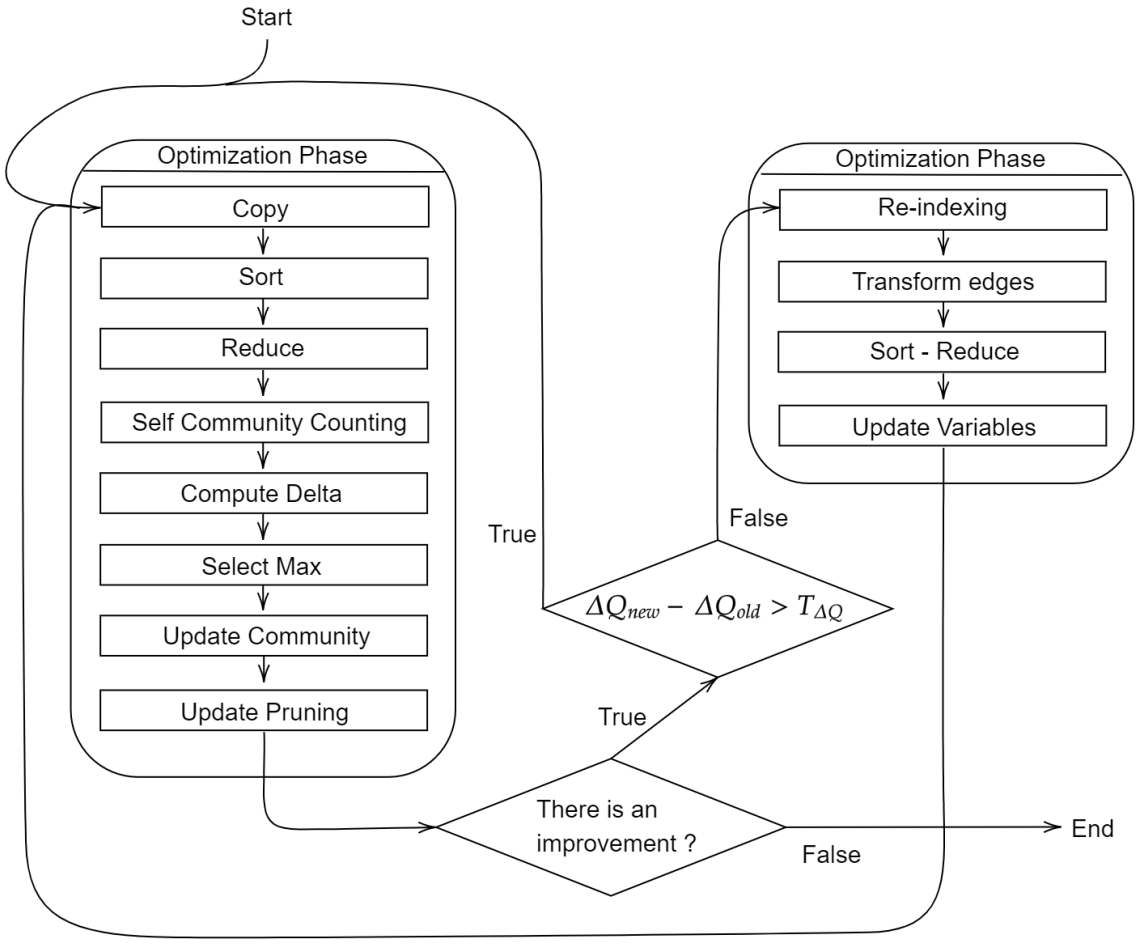


Figure 16: Schema of the PSR-Louvain algorithm.

1. **Copy sub-phase:** this algorithm implements firstly the pruning presented by Ozaki in the parallel behaviour [28]. Therefore, in the first step, we copy from the list of edges, all those that belongs to a node that we have consider in this iteration. To due this, we check on a support vector if the source nodes of the edges have a neighbour that has change community according to the criteria presented in Chapter 4.2. We use a support vector to check if the node matched the requirement: in position i there is a True if the nodes i matched the criteria in the previous iteration, False otherwise. At the first iteration all the nodes are considered. We exclude from the copy also the self-loops because we don't consider them in the computation of the various values of ΔQ. Besides, in this phase, we don't copy destination nodes j but we

substitute that values with the associated community c_j. Therefore, we obtain a list of tuples that contain the source node i, the community c_j related to the linked node j and the weight of the edge w.

2. **Sort sub-phase:** This algorithm uses a scheme of computation inspired by [19]. In the second phase, we sort the data obtained in the first phase. Given the list of tuples (i, c_j, w) created in the previous sub-phase, we sort it using the pair (i, c_j) as key. At the end of this phase, we have that all the tuples (i, c_j, w) with the same (i, c_j) are consecutive. Besides, all the tuples with the same source node i are consecutive.

3. **Reduce sub-phase:** in this step we perform a reduction by key, i.e. we sum up all consecutive values with the same key. We use as key the tuple (i, c_j): doing this, we obtain a unique tuple $(i, c_j,\ l_{i\to c_j})$ for each pair (i, c_j) where $l_{i\to c_j}$ is the sum of all the weights of edges that links the node i with a node in the community c_j. We need this value to calculate later the $\Delta Q_{i\to c_j}$.

4. **Self community counting sub-phase:** In this phase, we isolate all the tuples $(i, c_j, l_{i\to c_j})$ such that c_j is the actual community for the nodes i, i.e. c_i. We need to isolate these values $l_{i\to c_i}$ because we use it in the next phase to compute the various values of ΔQ (Eq. (22)).

5. **Compute Delta sub-phase:** Now we can calculate the $\Delta Q_{i\to c_j}$ for each tuple using the Eq. (22) , because we calculated all the $l_{i\to c_j}$ in the previous steps and we have the weights of the communities and the weights of the nodes in input. After the computation, we obtain a list of tuple $(i, c_j, \Delta Q_{i\to c})$.

6. **Select Max sub-phase:** Now we need to isolate the maximum $\Delta Q_{n\to c_j}$ for each node n. Considering that the vector is already sorted, we can perform this operation using another reduction but, in this case, we return only the maximum weight. We execute this operation using the nodes i as keys. After this step, we have exactly one tuple $(i, c_z, \Delta Q_{i\to c_z})$ for each nodes i where c_z is the community that gives the maximum increase in modularity $\Delta Q_{i\to c_z}$ if the node i is assigned to j.

7. **Update Community sub-phase:** In this step, we update the community for each node if the value of $\Delta Q_{i \to c_z}$ is greater than 0. We also update the related community weights: we remove the weight of the node from the previous community and we add it to the new one. These operations are implemented as atomic to avoid concurrent operations. We also keep track if each node changed or not its community. We remark that this vector is different respect to the array that we use in the first step. In the following step, we create the pruning vector from this one.

8. **Update Pruning sub-phase:** : To update the vector that handles the pruning criteria, we firstly set all its elements to zero. After that, a method takes each edge of the graph and check if the destination edge has changed its communities in the previous iteration: if this happened, the corresponding node value is set to True. This update operation is not atomic, because multiple threads can set only a True to the same position and there aren't conflict.

Afterwards, we compute the new modularity score and than we compare the obtained value with the old one: if the value is larger than a given threshold, we repeat these steps, otherwise we start the aggregation phase. We highlight that we can not add directly the various ΔQ obtained in the optimization step to the old modularity like the sequential algorithm, because all nodes change communities simultaneously and consequently this value is not reliable any more. The Algorithm 1 summarize the phase.

Algorithm 1 Prune-Sort-Reduce: Optimization phase

```
procedure optimizationPhase(Graph G, Community C)
    pruning = vector(True, n_nodes)
    old_delta = 0
    delta = modularity(G, C)
    while delta − old_delta < THRESHOLD do
        node_to_community = [ ]
        self_values = [ ]
                                                                  ▷ copy
        for each edge (i, j, w) in G in parallel do
            if pruning[i] == True and i != j then
                node_community.append(i, C[j], w)
                                                                  ▷ sort
        node_community.parallel_sort(by = source, community)
                                                                  ▷ reduce
        node_community.parallel_reduce( by = source, community,
                                        operation = sum)
                                                                  ▷ self-counting
        for each (i, cj, l) in node_community in parallel do
            if C(i) == cj then
                self_values[i] = l
                                                                  ▷ delta
        s = size(node_community)
        for z in [0, 1, ..., s] in parallel do
            node_community[z] = compute_delta(node_community[z],
                                              self_values,
                                              communities_weight,
                                              nodes_weight)
                                                                  ▷ max
        node_community.parallel_reduce( by = source,
                                operation = max)
                                                                  ▷ update community
        is_change = vector(False, n_nodes)
        for each (i, cz, delta) in node_to_community in parallel do
            if delta > 0 then
                atomicAdd(communites_weight[C[i]], −nodes_weight[i])
                atomicAdd(communites_weight[cz], nodes_weight[i])
                C[i] = cz
                is_change[i] = True
                                                                  ▷ update pruning
        for each edge (i, j, w) in G in parallel do
            if is_change[j] == True then
                pruning[i] = True
        old_delta = delta
        delta = modularity(G, C)
```

The aggregation phase uses several similar concepts presented previously, and we can divide it into four sub-phases in which the operations are executed in parallel:

1. **Re-indexing communities sub-phase:** in the first phase, before the graph contraction, we assign a new id to the communities. Actually, we have only certain communities associated to the nodes respect to the initial configuration: for example, if a nodes i change community from c_1 to c_2 in the first iteration of the optimization phase, no nodes are assigned to c_1 after the update and no nodes can select the communities c_1 from that moment. This cause a useless waste of memory if we continue to keep all those unused values in the community weight. For this reason, we need to create a map to rearrange the communities index. First, we create a support vector such that at the position c there is a 1 if the community c has a weight greater than 0 (i.e. there is at least one node assigned to this community). Then we perform a prefix sum on this vector: in this way at the position c there is the new index incremented by one for the community c (please note: incremented by one because we counting from zero). We remark that even if the empty communities are still mapped with this method and have the same indexes of the community at the preceding position respect them, we doesn't have any conflict because we doesn't use these entry. When this renumbering map is ready, we start the next phase.

2. **Transform edges sub-phase:** In this step, all the pairs of edges (i, j) of the original graph are transformed in the pair (c_i, c_j) where c_i (c_j) is the community associated to i (j). In this phase, we also apply the map to renumber the communities that we create in the previous step.

3. **Sort-Reduce sub-phase:** In this phase we sort all the edges (c_i, c_j, w) using as a key for the sorting the pair (c_i, c_j). After this, we reduce the edges vector still using as a key (c_i, c_j). After this step we have contract the graph summing up all the edges that lay between two communities.

4. **Update variables sub-phase:** In the last step, we update all the support value in the graph object, like the number of the nodes, the number of edges,

the nodes weights. We also reset the communities object reordering the communities weight according to the re-indexing map and assigning each node to a community composed only by itself.

The Algorithm 2 summarizes this phase. The PSR-Louvain continues to alternate this two phases until we can not have further improvement in the modularity update. In this version of the algorithm, we keep only the best result to not occupy several device memory, but it is possible trivially save the intermediate result adding a step that save the clustering results after the re-indexing sub-phase (in this way we have consistent indexing among the dendrogram).

Algorithm 2 Prune-Sort-Reduce: Aggregation phase

```
procedure AGGREGATIONPHASE(Graph G, Community C)
                                                        ▷ re-indexing
    for i in size(communities_weight) in parallel do
        if communities_weight[i] == 0) then
            map[i] = 0
        else
            map[i] = 1
    map.prefix_sum()
                                                        ▷ transform
    for each edge (i, j, w) in G in parallel do
        Substitute i, j with map[C[i]], map[C[j]]
                                                        ▷ sort-reduce
    G.edges.parallel_sort(by = "source, community")
    G.edges.parallel_reduce(by = "source, community",
                            operation = "sum")
                                                        ▷ update
    G.update()
    C.update(G)
```

5.2 PH-Louvain

The second version of the parallel Louvain algorithm is quite similar to the previous pruning approach, but uses a different way to aggregate the weights of edges that link a node to the same community: we use a special global hashmap instead of sorted vector. Using a map to accumulate some values by its key is a standard approach to solve this problem because the map allows to retrieve and insert an object in $O(1)$ time. To obtain this performance, the hashmap uses a function named hash function to dispose at random the objects in the memory. This creates a problem on the GPU because uncoalesced memory accesses is an order of magnitude slower than sequential memory accesses. To overcome this problem this map uses a system of open-addressing based on the cuckoo hashing: this type of map is the one that performs better on the GPUs [18], thanks to a simple and efficient management of the conflicts. This map uses 64 bits for the key and 32 bits for the value. We choose to use 64 bits for the key because we need to store a pair of 32 bits keys (the pair $(node, community)$ in the optimization phase and the $(community, community)$ pair in the aggregation phase). This map has r different hash function, each one associated to an id r_i where $i \in [0, r-1]$. When we insert a new pair key-value (k, v), we use the hash function with id r_0 to compute the position of the new key v: if the slot is empty, we add the key and the value, we use the next function r_1 otherwise. We continue to search a empty slot following this schema: if r_i is not empty, we retry to insert the pair with function r_{i+1}. If all the function fails to insert the new pair, we raise an error.

The main difference between this map and the classic cuckoo hashing is that the original version of the map when we try to insert an entry in a position that is already filled, the map remove the old pair key-value, insert the new one and try to find a new position for the old value using another hash function. In our map, we don't "kick out" the old key when we have a conflict in order to find a new memory address for it, but we hash with a different function the pair that we have to insert: this because to make a classic cuckoo hashing, we need a set of atomic operations for 128bits to do the "kick out" operation in parallel without generating race condition. This type of atomic operation in CUDA can be done only on variable up to 64

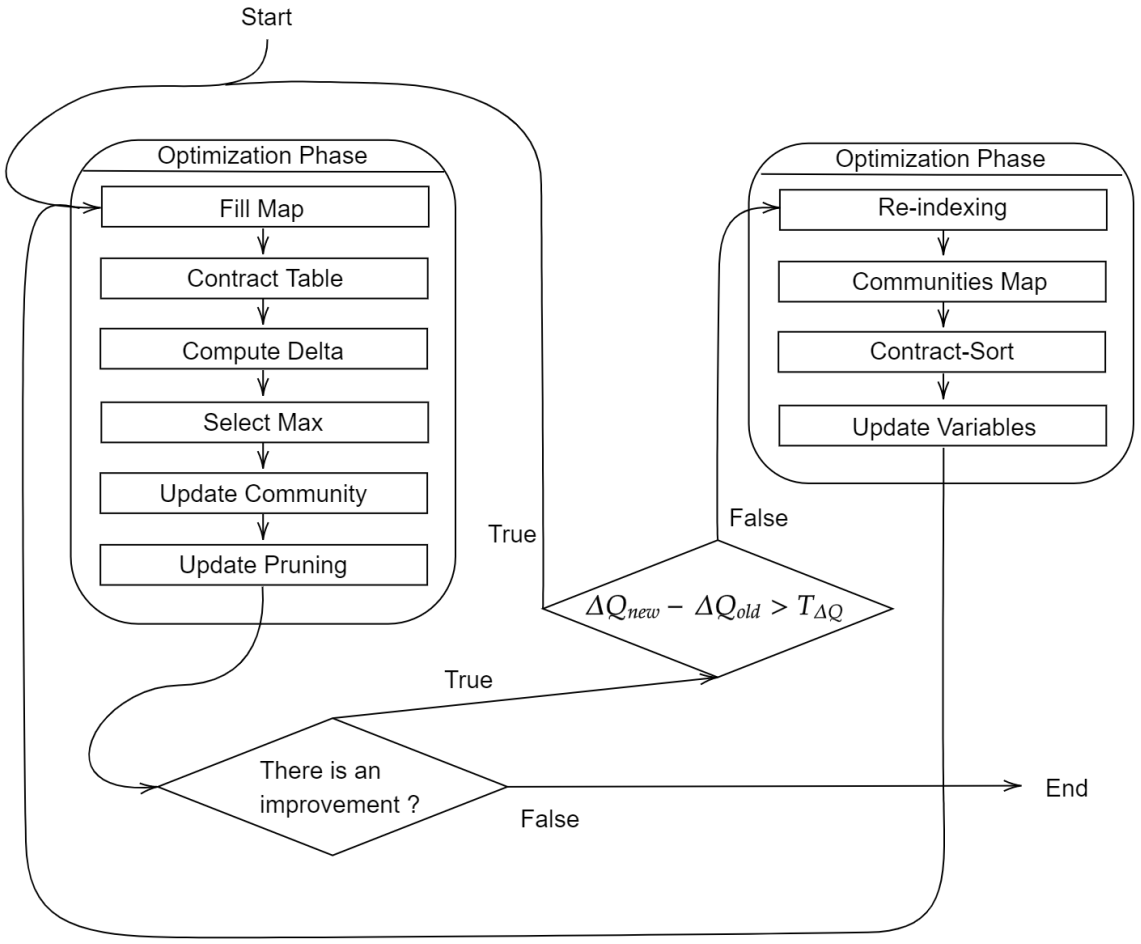


Figure 17: Schema of the PH-Louvain algorithm

bits. Besides, this map has another special feature: if we insert a pair (k, v) and there is already an entry in the table with a key k, the map automatically sum the values v to the one stored in the map. Indeed, we use this map only to aggregate values, and for this reason, we design it to this operation as fast as possible. The last feature that we add to this table is the contract table operation: this methods the map in pair of contiguous vector re-arranging the memory, in order to allow us to access to each pair key-value sequentially. After this operation, we can not get the entry using the hash function, because the memory is re-organized. We use this operation before the computation of ΔQ to increase the performance and still working considering the edges $(nodes, communities)$ independently. Now we present the algorithm. At the start of the optimization phase we have each edge represented

as a tuple (i, j, w), a the weight of each node, a map that associates each nodes with the corresponding communities and the the total weight of each communities like the previous algorithm. We divide the optimization phase into six sub-phases:

1. **Fill Map sub-phase:** in the first step, for each edge (i, j, w), we insert into the map the tuple (n, c_j, w) where c_j is the community associated to the destination nodes j. We use as a key the pair (n, c_j). We consider only the tuple that has the source node that matching the criteria presented in Chapter 4.2. We use a support vector identical to the one presented in the previous algorithm. We also exclude all the self-loops in this phase. At the end of this step we obtain a map where every non-empty entry has the form $(i, c_i, l_{i\to c_j})$. The map also isolate in a different vector each values $l_{i\to c_i}$ such that c_i is the community of the nodes i is stored at position i. This operation was made because in the delta step, after the contraction, we can not get any more the position of the entry using the hash function, and we need a method to get the nodes communities weights quickly.

2. **Contract Table sub-phase:** in the first step, using a map to calculate $l_{i\to c_j}$ allows accessing to the memory address in $O(1)$ time for each edge. But to compute the relative delta and the maximum, we have to access the memory using an uncoalesced memory access pattern. In addition, we have no idea of how and which community are in the neighbourhood of a given node: many application allocates a thread for each edge, transform the edge destination from nodes to community and then check the maximum. This approach lay to check multiple time the score of the community c if two neighbours of the nodes n are in c. To overcome this problem, we contract the table: the vector that composed the table are re-organized in order to permit sequential access, without empty slots between the entries. After this operation, the map becomes a vector containing the tuples $(i, c_j, l_{i\to c_j})$. The support vector with the sum of edges of the node that connects it with another in the same community is not re-arranged by this operation.

3. **Compute Delta sub-phase:** now we can compute the $\Delta Q_{i\to c_j}$ for each tuple

$(i, c_j, l_{i \to c_j})$ using the Eq. (22). The result overwrites the last value in the tuple (we doesn't need that value any more). To get the sum of edges that connect the nodes to its actual community, we use the vector created in the first step.

4. **Select Max sub-phase:** now we have a vector of tuple $(i, c_j, \Delta Q_{i \to c_j})$. Now we have to select the pair $(c_i, \Delta Q_{i \to c_j})$ for each node i such that $\Delta Q_{i \to c_j}$ is maximum. We can not use as the previous algorithm a reduce operation because we haven't a sorted array and the sorting operation is too expansive. Instead, for each tuple, we check if the value $\Delta Q_{i \to c_j}$ is greater compered to the one of the tuple saved in a support array at the position i: if it is, we substitute the old value with the new one, we do nothing otherwise. To avoid race condition in the insert, these operation are executed atomically. At the end of this step, we have exactly one tuple $(i, c_z, \Delta Q_{n \to c_x)}$ for each node i, like the previous algorithm.

5. **Update Community sub-phase:** This phase update the community just like the one presented in the Prune-Sort-Reduce version (5.1, optimization sub-phase 7).

6. **Update Pruning sub-phase:** This phase create the array with the pruning information just like the one presented in the Prune-Sort-Reduce version (5.1, optimization sub-phase 8).

The Algorithm 3 summarizes this phase.

Algorithm 3 Hashmap: Optimization phase

procedure OPTIMIZATIONPHASE(Graph G, Community C)
$pruning = vector(True, n\_nodes)$
$old\_delta = 0$
$delta = modularity(G, C)$

while $delta - old\_delta <$ THRESHOLD **do**

▷ fill map

for each edge (i, j, w) in G **in parallel do**
if $pruning[i] == True$ and i != j **then**
$hashmap.insert(i, C[j], w)$

▷ contract

$list = hashmap.contract()$

▷ delta

for z in size(list) **in parallel do**
$list[z] = compute\_delta(list[z], hashmap.self\_values,$
$communities\_weight, nodes\_weight)$

▷ max

$result = [\,]$
for each $(i, cj, delta)$ in list **in parallel do**
START OF THE ATOMIC BLOCK
$(i, cj_{old}, delta_{old}) = result[z]$
if $delta > delta_{old}$ **then**
$result[z] = (i, cj, delta)$
END OF THE ATOMIC BLOCK
$list = result$

▷ update community

$is\_change = vector(False, n_nodes)$
for each $(i, cz, delta)$ in list **in parallel do**
if $delta > 0$ **then**
$atomicAdd(communites\_weight[C[i]], -nodes\_weight[i])$
$atomicAdd(communites\_weight[cz], nodes\_weight[i])$
$C[i] = cz$
$is\_change[i] = True$

▷ update pruning

for each edge (i, j, w) in G **in parallel do**
if $is\_change[j] == True$ **then**
$pruning[i] = True$

$old\_delta = delta$
$delta = modularity(G, C)$

We continue to execute this step until the difference in modularity between the configuration drops below a given threshold. The consideration of the computing of ΔQ in parallel behaviour presented in the previous chapter is still valid. The aggregation phase of this algorithm use once again the map to aggregate, but the key in this context is composed of $(community_source, community_destination)$. We can divide this phase in four sub-phase like the previous version:

1. **Re-indexing communities sub-phase:** In this sub-phase we associate each community to a new id to reduce the waste of memory. This sub-phase is identical to the sub-phase with the same name presented in the PSR-Louvain description (5.1, aggregation sub-phase 1).

2. **Communities map sub-phase:** In this step, all the tuple of edges (i, j, w) of the original graph are inserted in a hash table. Before the insertion, we transform each entry in the tuple (r_i, r_j, w) where r_i is an index of the community associated with i after the remapping. We use as key the pair (r_i, r_j). At the end of this step, we have each edge of the new graph, because we sum up all the edges that lay between two communities.

3. **Contract-sort sub-phase:** At the beginning of this phase, we have a map containing all the edges of the new graph. However, the graph object store the edges information using three ordered vectors, so we have to re-organize the information stored in the unordered and uncoalesced map. To do this, we use the contract operation to transform the map in a vector of tuple (c_i, c_j, tot) and then we sort the arrays according to the order of the first one. Finally, we have the updated graph.

4. **Update variables sub-phase:** This phase update the new graph and the related communities object just like the one presented in the Prune-Sort-Reduce version (5.1, aggregation sub-phase 4).

The Algorithm 4 summarize this phase. Like the previous algorithm, this one continues to alternate the two main phases until no further improvement in the modularity update can be obtained.

Algorithm 4 Hashmap: Aggregation phase

```
procedure AGGREGATIONPHASE(Graph G, Community C)
                                                        ▷ re-indexing
    for i in size(communities_weight) in parallel do
        if communities_weight[i] == 0) then
            map[i] = 0
        else
            map[i] = 1
    map.prefix_sum()

                                                        ▷ map
    for each edge (i, j, w) in G in parallel do
        hashmap.insert(map[C[i]], map[C[j]], w)
                                                        ▷ contract-sort
    G.edges = hashmap.contract()
    G.edges.parallel_sort(by = "source, community")

                                                        ▷ update
    G.update()
    C.update(G)
```

5.3 Speed-up the First Iteration in the Optimization Phase

In this chapter, we present an optimization technique that we add into our code to speed-up the first iteration of each optimization phase. We include this method in both versions of the algorithm presented previously. We focus this presentation on the Prune-Sort-Reduce version, even if the concept that allows us to optimize the algorithm is still present in the Hashmap version. At the beginning and also after each aggregation phase, we notice that we have a configuration in which each node is assigned to each self-community, i.e., each node i is assigned to the community c_i and it is the only node assigned to it. In this configuration, when in the copy sub-phase we transform each edge (i, j) in the pair (i, c_j) where c_j is the community of the node j, we obtain the same original pair, because the index of c_j is equal to j. In addition, considering that each node is assigned to a different community, we don't need the sort and reduce sub-phases, because the pairs (i, j) are already sorted by the construction of the graph object ()we need a sorted vector for the select max sub-phase) and the reduction is useless, because all the pairs have a different composite key (i, j). Also the self-counting sub-phase is useless, because

no edge that isn't a self-loop lay a node in the same community and during the copy sub-phase we excluding the self-loop from the computation. Considering all these facts, we choose to remove in the first iteration this three sub-phases and also to avoid the useless transformation in the copy sub-phase. For the Hashmap version of the algorithm, we still use this optimization based on the other version because we use the hashmap to aggregate different edges and this aggregation is no longer necessary.

5.4 Data structure and implementation details

The two main structures that we need to represent are the original graph and the community structure. Commonly a graph $G(V, E)$ is represented using its adjacency matrix: each node is associated with an incremental id in the range $(0, n-1)$ where n is the number of nodes. To retrieve the weights of an edge between two nodes, we look to the values stored at the position (id_1, id_2). As we say in the chapter 3.1, the community detection techniques are effective if executed on sparse graphs. Therefore, if we use a matrix to represent a sparse graph, this matrix will have a lot of zeros. Even if this pattern allows to retrieve the weight of an edge in constant time, this data structure isn't suitable for a problem that involves millions of data because we need too much memory to allocate this matrix and it has several unused values. Therefore, we have the necessity of compress the adjacency matrix. To do this, we choose to represent it using a coordinate list (often referred to as COO). We have three vectors `edges_source`, `edges_destination` and `weights` that contain respectively the ids of the rows, the ids of the columns and the values. The standard modularity optimization is computed on undirected graphs: this means that the adjacency matrix is symmetric and we can store in the COO list only the upper triangle of the matrix and we still have all the edges represented. Despite this observation, we store all the values of the adjacency matrix because we need those repeated values in these algorithms when we transform the destination node in the destination community. These vectors are also sorted by the pair $(source, destination)$ and there is a vector named `neighboorhood_sum` of length n in which at position i there is the starting position of the edges that have source

i in `edge_source`. Thanks to this vector, we can retrieve fast all the neighbour of a given node. These particular COO lists are also known as compressed neighbour lists. In the graph object, we also store a vector named `tot_weight_per_nodes` that associate each node i to its degree, the total degree of the graph, the number of nodes and the number of the edges. We use `thrust::device_vector` to store all this data on the GPUs memory. In summary, the graph object is the following:

```
struct GraphDevice {
  unsigned int n_nodes;
  unsigned long n_links;
  double total_weight;

  thrust::device_vector<unsigned> tot_weight_per_nodes;
  thrust::device_vector<unsigned int> neighbourhood_sum;

  thrust::device_vector<unsigned int> edge_source;
  thrust::device_vector<unsigned int> edge_destination;
  thrust::device_vector<unsigned int> weights;
}
```

We design the community object so that it contains the graph associated with it. We notice that the maximum number of different communities is pair to the total number of nodes: this is also the configuration at the beginning of the algorithm. Considering this, we choose to identify each community with an incremental id in the range $(0, n-1)$, like we do previously with the nodes. Therefore, in the community object, we have a vector named `communities` of length n in which in the position i there is the id of the community of the node i. Besides, there is a vector of size n that associate each community to its total weight. In summary, the community object is the following:

```
struct Community {
  GraphDevice graph;

  thrust::device_vector<unsigned int> communities;
  thrust::device_vector<double> communities_weight;
}
```

After the initial parsing, we use only the device memory to store the data. The GPU memory is related to the machine and is usually smaller than a RAM. To reduce the memory used simultaneously during the optimization phase, the Prune-Sort-Reduce algorithm divides the edges in buckets of fixed size. The algorithm executes the sub-phases from the first to sixth on a bucket, stores the results and start the execution on another bucket. When all buckets are considered, we execute the last two updating sub-phases. This reduce the required memory used to store the data from $2 * m$ (used to store two vector that can store all the nodes: one for the copy and one to store the reduced values) to $2 * size(bucket) + n$ (two vector of the bucket size for the copy and the reduce operation and a vector to store the best result for each node). In order to compute the right maximum values of ΔQ for each node, we split the edges into buckets such that if there is an edge with source node n in the bucket, also all the other edges that belong to n are included. To perform this, we use the `neighbourhood_sum` vector to select the right range. The Hashmap version implements a similar logic, performing repeatedly the the first four sub-phases on different buckets, and then eventually update the communities and the pruning vector when all buckets are processed. We use buckets of maximum size equals to 50 000 000.
Both algorithms implements also the minimum label heuristic (Chapter 4.3): we change communities only if the id of the new communities is lower than the old one for communities composed only by a single node to prevent the simultaneous swap. We also select the communities with the lowest id when we have multiple communities with the greatest ΔQ for the same node: this method leads to a faster convergence.

In the algorithms, each parallel sub-phase presented for both algorithm takes as an input a list of element on which execute its computation, and we assign to each thread one of this element in our kernels. We also use the `thrust` library to perform the sorting, the transforming, the reducing and the prefix sum operations. When we need to consider the edges vector as a unique tuple, we use a `zip_iterator` to handle the data correctly. The hashmap is stored in the device global memory and it is composed by two `thrust::device_vector`: the first one, used for the keys,

contains a `unsigned long long`; the second one `float`, used for the value, contains `float`. We choose to use a
`unsigned long long` because in this way we can perform atomic operation on the composite key. The contract table operation is made quickly using the function `thrust::remove_if` that removes from the vector every element x that satisfies a predicate and then contract the vector. The predicate that we use check if the memory slot in the vectors is empty.

In the Select max sub-phases of the Hashmap version, to avoid race condition caused by the two atomic operation that we have using considering independently each community c and the associated $\Delta Q_{i \to c}$ (`atomicMax` on the value and possibly `atomicCAS` (compare and swap) for the associated community), in the implementation we transform this of 32 bits variable in a unique word of 64 bit. The half most significant bytes are filled by the value, the other part by the community. Thanks to this, we can consider it as a `unsigned long long` and we can use an unique atomicMax to substitute both the values.

6 Performance and Analysis

In this chapter we present the results obtained by these two algorithm. We use 13 dataset to test the algorithm, and we analyze the performance globally on the entire algorithm but also locally on specific phases and sub-phases in order to highlight the point of strengths and the weakness of both the algorithm. In this chapter we firstly present the datasets on which we have executed our test. Then we give an overlook on the results obtained that are useful for the subsequent analysis which, in turn, we dived in two parts: in the first part we analyze the contribution of the pruning approach on each algorithm; in the second part we make a comparison between the two algorithms, in order to enhance when one algorithm performs better than the other. This consideration are on base to the adaptive algorithm that we present in the next chapter. It implements the logic of the two algorithm and select to use the best basing on the situation, so the information that we have collect in this part are foundamental for the design choice that we make.

6.1 Datasets

In this section we present the 13 dataset used in this thesis. These datasets coming from three main sources: the Stanford Stanford Large Network Dataset Collection (also known as SNAP) [21], the Florida Sparse Matrix Collection [17] and the the Koblenz Network Collection (also known as KONECT) [20]. In the Table 1 there are a briefly presentation of the dataset and in Figure 18 there are the degree distribution of the graphs. We point out that even if all this graph are unweighted, some of it are directed: during the parsing phase we doesn't keep the "directness" of the graph, and we treat it as a undirected one as required from the algorithm. In addition, if in the original graph there are some repeated edges, we consider it once. In the table 1 the number of edges are those obtained after this parsing, ordered by increasing edges number. Now we present the datasets:

- **coPapersDBLP:** this graph model the citation and co-author network from the DBLP - Digital Bibliography and Library Project. Each node represent an author and each edge a citation.

Datasets			
Name	Source	Number of nodes	Number of edges
coPapersDBLP	Florida	540 486	15 245 729
patentCite	KONECT	3 774 768	16 518 947
packing-500x100x100-b050	Florida	2 145 839	17 488 243
soc-pokec-relationship	SNAP	1 632 803	22 301 964
delaunay_n23	Florida	8 388 608	25 165 784
soc-LiveJournal1	SNAP	4 847 571	43 369 619
wikipedia_link_ja	KONECT	1 610 638	56 237 763
hollywood-2009	Florida	1 139 905	57 515 616
wikipedia_link_it	KONECT	1 865 965	68 049 979
wikipedia_link_fr	KONECT	3 023 165	83 472 152
com-orkut	SNAP	3 072 441	117 185 083
wikipedia_en(dbpedia)	KONECT	18 268 991	126 890 209
indochina-2004	Florida	7 414 768	153 487 303

Table 1: Datasets overview

- **patentCite:** This is the citation network of patents registered with the United States Patent and Trademark Office. Each node is a patent, and a directed edge represents a patent and an edge represents a citation. The network contain loops. This graph is also directed.

- **packing-500x100x100-b050:** this is a synthetic graph from numerical simulation.

- **soc-pokec:** Pokec is the most popular on-line social network in Slovakia. Pokec has been provided for more than 10 years and connects more than 1.6 million people. This dataset map the relationship between people.

- **delaunay_n23:** given a random set of point, in this graph represent a Delaunay triangulations of them.

- **soc-LiveJournal1**: LiveJournal is a free on-line community with almost 10 million members; a significant fraction of these members are highly active. LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong. This graph model these friendship relations.

- **wikipedia_link_jp:** This network consists of the wikilinks of the Wikipedia

in the Japanese language (.ja). Nodes are Wikipedia articles, and directed edges are hyperlinks. Only pages in the article namespace are included. This graph is directed and some self-loop is possible.

- **hollywood-2009**: The graph of movie actors. Vertices are actors, and two actors are joined by an edge whenever they appeared in a movie together. The data date back to 2009.

- **wikipedia_link_it:** This network consists of the wikilinks of the Wikipedia in the Italian language (it). Nodes are Wikipedia articles, and directed edges are hyperlinks. Only pages in the article namespace are included. This graph is directed and some self-loop is possible.

- **wikipedia_link_fr:** This network consists of the wikilinks of the Wikipedia in the French language (.fr). Nodes are Wikipedia articles, and directed edges are hyperlinks. Only pages in the article namespace are included. This graph is directed and some self-loop is possible.

- **com-orkut:** Orkut is a social-network where users form friendship each other: the nodes represent the users and the edges the friendship between them.

- **wikipedia_en (dbpedia)**: The network is the hyperlink network of Wikipedia, as extracted in DBpedia. Nodes are pages in Wikipedia and edges correspond to hyperlinks (also known as wikilinks). The edges correspond to the relationships in DBpedia. Network info. The original graph is directed and multiple edges are possible.

- **indochina-2004**: A fairly large crawl of the country domains of Indochina performed for the Nagaoka University of Technology. This is a directed graph and each node represent a site and each edge a link from one site to another.

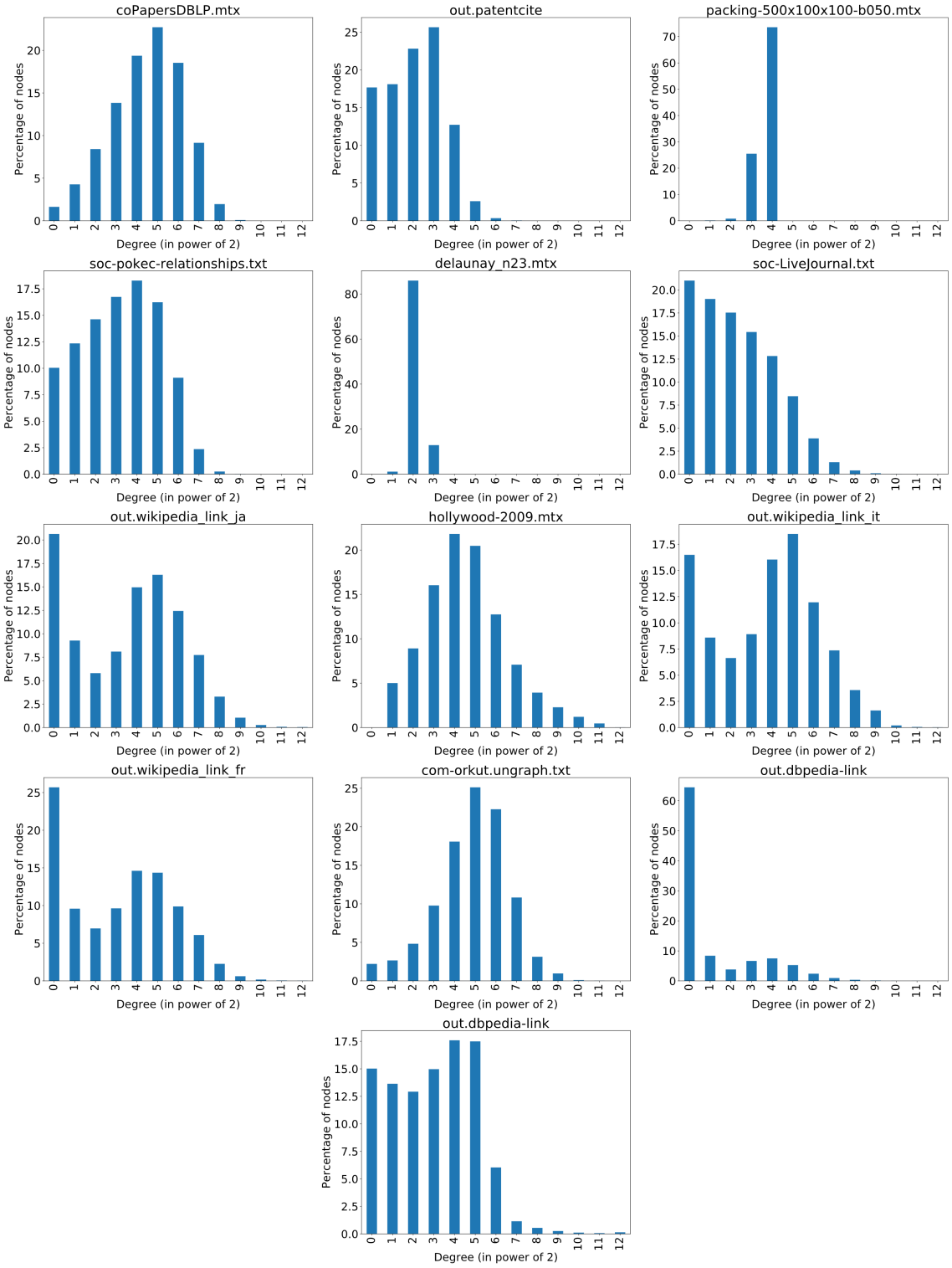


Figure 18: Degree distribution in the datasets: we divide the nodes in ordered classes where in the class i there are all the nodes with degree in range $[2^i, 2^{i+1})$. On the x-axes there are these classes; on the y axes there is the percentage of nodes assigned to them.

6.2 Results Overview

In this section we present an overview of the obtained results and we make some general consideration about it. Some more insights about the first optimization phase and about the aggregation phase are presented next and we make some consideration in light of what we present in this section.

The algorithms was run on a Ubuntu 18.04.4 LTS machine with a 2.10GHz Intel(R) Xeon(R) Silver 4110 CPU, a TITAN Xp GPU with 12 GB of memory and CUDA 10.2. To run our code we need a Nvidia GPU with a compute capability ≥ 3.5 due to the 64 bits `atomicMax` operation used in PH-Louvain. This GPU have a compute capability 6.1, so it comply with the technical specification. We remark that we keep executing our optimization phase until the value of ΔQ between the various iteration is greater than a threshold t. Both our parallel version in the test uses $t = 0.01$.

Modularity Score			
Graph	Sequential	PSR-Louvain	PH-Louvain
coPapersDBLP	0.8490	0.8544	0.8544
patentCite	0.8095	0.7911	0.7878
packing-500x100x100-b050	0.9353	0.9434	0.9416
soc-pokec-relationship	0.6837	0.6934	0.6852
delaunay_n23	0.9881	0.9856	0.9857
soc-LiveJournal1	0.7251	0.7491	0.7482
wikipedia_link_ja	0.5928	0.5690	0.5724
hollywood-2009	0.7511	0.7542	0.7542
wikipedia_link_it	0.7221	0.7190	0.7196
wikipedia_link_fr	0.6777	0.6834	0.6871
com-orkut	0.6545	0.6613	0.6629
wikipedia_en(dbpedia)	0.6629	0.6612	0.6618
indochina-2004	0.9638	0.9632	0.9632

Table 2: Modularity result

First of all, we analyze the modularity score obtained by the two algorithms respect to the sequential version as presented in [14] to check the correctness of the our algorithm. In the Table 2 are exposed the obtained results. We notice that both the score of the algorithm are in pair between them and also with the sequential version for all the graphs, and in some case we obtain also a better result in the parallel

Execution Times			
Graph	Sequential	PSR-Louvain	PH-Louvain
coPapersDBLP	11 906.89	419.59	412.79
patentCite	88 620.13	1 796.88	2 555.14
packing-500x100x100-b050	13 746.14	1 045.25	1 090.03
soc-pokec-relationship	30 162.70	1 843.95	2 186.81
delaunay_n23	44 392.42	1 020.23	1 319.22
soc-LiveJournal1	77 225.64	2 187.45	2 677.51
wikipedia_link_ja	76 816.01	2 654.88	2 305.11
hollywood-2009	52 306.71	2 092.09	1 758.27
wikipedia_link_it	82 599.92	3 875.04	2 732.99
wikipedia_link_fr	115 977.81	3 910.95	3 273.91
com-orkut	193 835.34	7 566.90	7 484.10
wikipedia_en(dbpedia)	300 431.38	5 287.65	6 464.03
indochina-2004	113 195.87	2 899.50	2 303.52

Table 3: Execution Time in milliseconds

implementations respect to the sequential version. This may be due to the parallel optimization, changing all the communities assigned to the vertices simultaneously, can avoid some local maxima.

In the Table 3 there are the execution time of the two algorithms respect to the sequential version. We notice a big improvement in the performance for both the algorithm respect to the sequential version: we obtain a speed-up in range of a variable factor from 12 to 56 for our two algorithms. Besides, we notice that, according to the graph analyzed, the two algorithms obtain different performance. In some case one approach outperforms the other of even more than one second. In the following sections we analyze in details this two algorithm to discover the motivation of this different performances.

6.3 Pruning analysis

In this section, we analyze the effectiveness of the pruning approach: we focus our research on the first optimization phase. As we said previously, the first optimization phase is the most time-requiring phase, consuming about 80% of the time [22], so the pruning approach should increase the performance especially in this phase. For this reason, we concentrate our analysis of this technique over it. First of all we present the percentage of the edges that we analyze in each iteration of the optimization

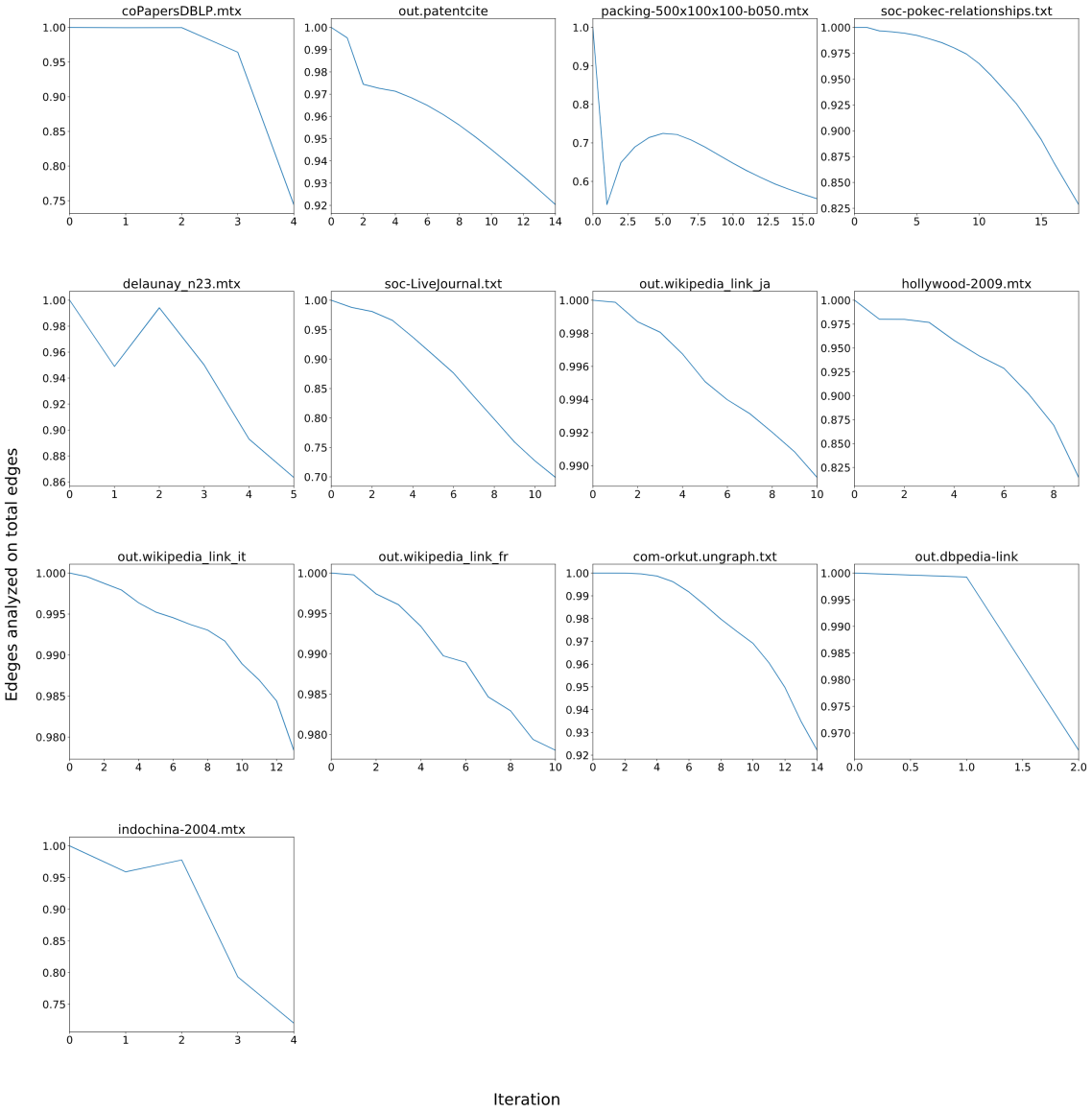


Figure 19: Percentage of edge analyzed in the first optimization phase.

phase (Figure 19). The number of the edges analyze are quite similar for both the algorithms. Exluding certain fluctuation at the earliest stage (we remark also that the two graph with the highest noise are two syntetic ones, i.e. packing-500x100x100-b050 and delaunay_n23), we notice that the portion of the edges analyzed tend to decrease iteration by iteration. The percentage of reduction highly depends on the graph exterminated: we have the smallest reduction for the wikipedia_link_ja (only the $\sim$ %2 of the total are excluded in the last iteration); instead, in the graph packing-500x100x100-b050, we have run the optimization only on the $\sim$ %50 of the edges of the graph in last iteration. Our study on the dataset doesn't find any direct

correlation between the decrease of the number of the edges analyzed and the degree distribution of the nodes presented in Figure 18.

6.3.1 PSR-Louvain analysis

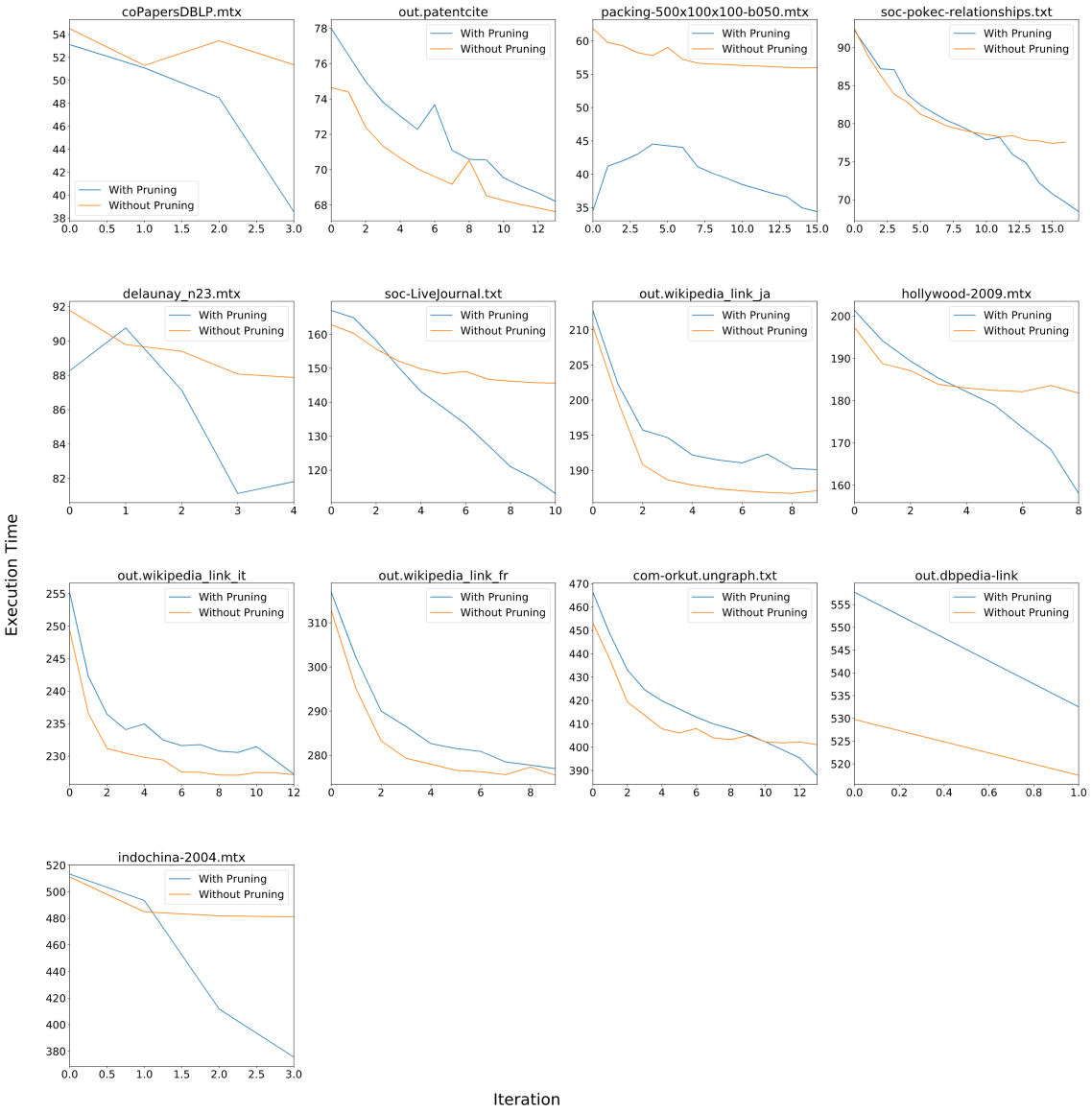


Figure 20: Comparison of the execution time between the Prune-Sort-Reduce (in blue) and a version of comparison without the Pruning approach (in orange) for each execution of the first optimization phase, excluding the first one.

Now we evaluate the impact in terms of time of the pruning approach in the PSR-Louvain algorithm: to do this, we create a comparison version of the algorithm from the presented one. This version doesn't prune the node in the Copy sub-phase and

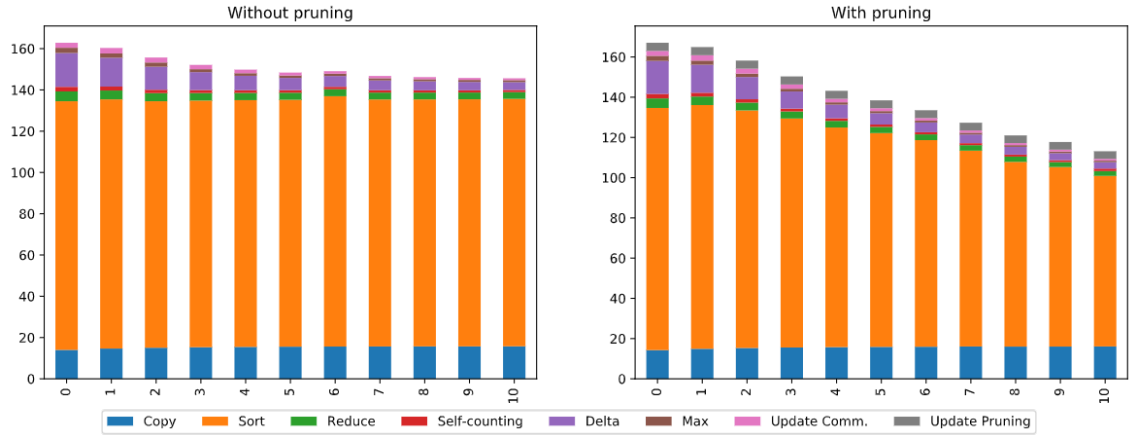


Figure 21: Comparison of the execution time between the Prune-Sort-Reduce (on the left) and a version of comparison without the Pruning approach (on the pruning) for each execution of the first iteration phase in the LiveJournal1 dataset. We highlight the contribution of each sub-phase in terms of time.

doesn't collect the data used to create the support pruning array in the last two sub-phases (the first one only update the community assign to each node; the second one is skipped). The execution times of each iteration of the first phase are illustrated in the Figure 20. We exclude from this graphic the first iteration because this one doesn't make the same step of the other due to its special optimization (Chapter 5.3). We notice that the reduction in terms of times in the pruning version is proportional to the reduction of edges analyzed: in the graph in which the reduction of the edges analyzed highly decrease, the pruning version outperform the the standard version; in the graph in which the reduction in terms of edges are small, the execution times are quite similar: the pruning algorithm is slower up to $\sim$ 15 ms at iteration (in dbpedia-link) due to the fact that the computation of the pruning information requires some extra time but the reduction that leads to is smaller. In general we see that the ratio between execution times of the pruning version and the non-pruning one is in pair with the number of nodes pruned in each iteration. We notice also that the execution times decrease iteration by iteration even for the version without the pruning: this is caused by the reduction in terms of the time of the update subphase, as we can see in the 21 is illustrated the contribution of each sub-phase in terms of time to for each iteration for the pruning version and the standard in the LiveJournal1 dataset. We notice that the Sort sub-phase is the most consuming one

Figure 22: Contribution of each sub-phase in terms of time for each iteration of the first optimization phase.

and this behaviour is similar also in the other graph: this sub-phase at least 50% of the time, reaching peek of even more than 80% of the time in the first and in the last graph (Figure 22). From the Figure 21, we notice also that the pruning on the data in input have a direct effect on the sorting phase and the decreasing in terms of time is proportional to the number of the edges pruned (as we can see by comparing Figure 22 and Figure 19). The pruning update has a small impact to the total execution time. In conclusion, even if the pruning approach isn't always effective and can introduce a little overhead, the gain in terms of time that we obtain when

we prune a portion of the edges are high, therefore adding the pruning approach is a valid optimization techniques for the algorithm that use a sort-reduce pattern to aggregate.

6.3.2 Hashmap algorithm analysis

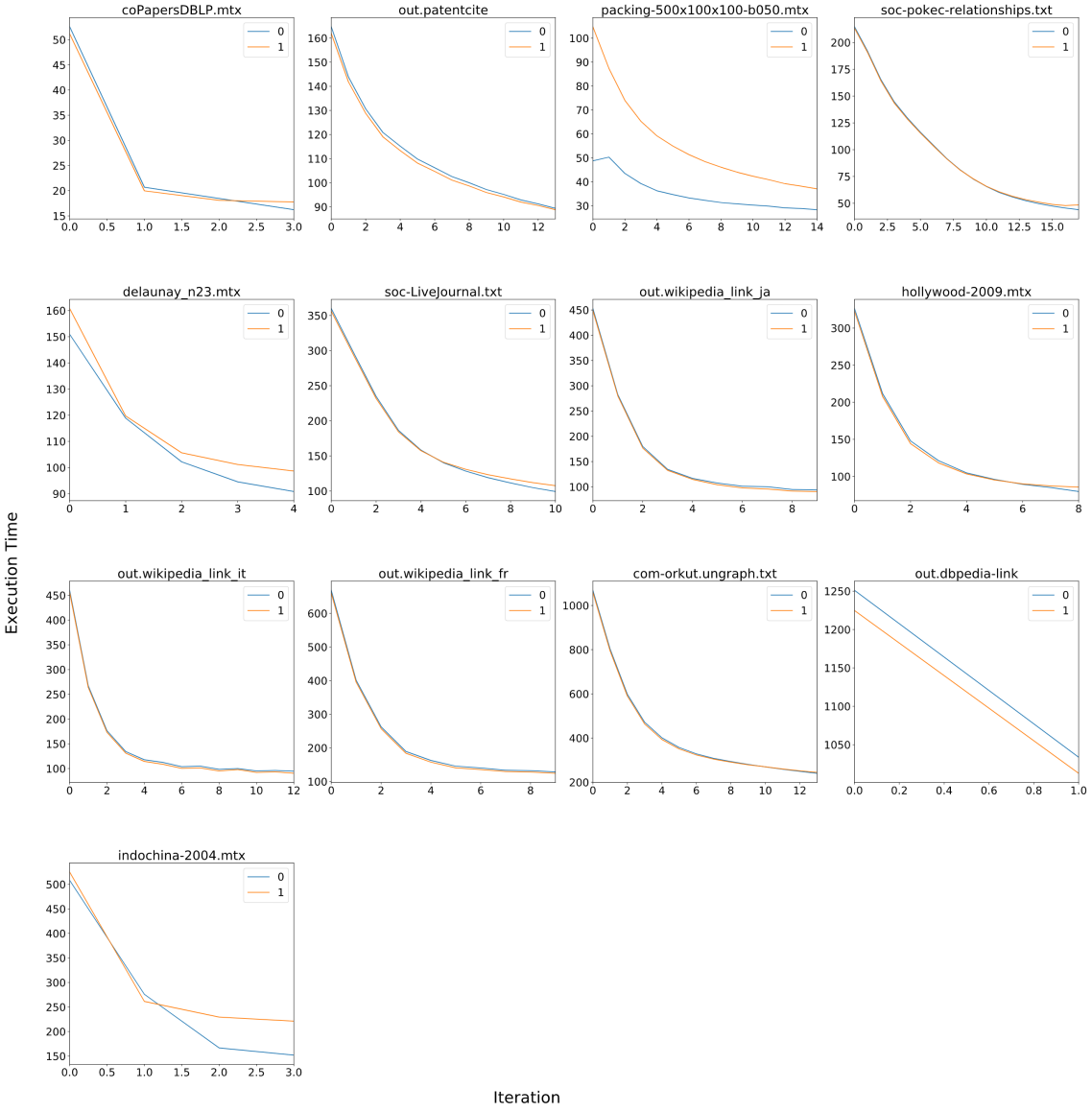


Figure 23: Comparison of the execution time between the Hashmap (in blue) and a version of comparison without the pruning approach (in orange) for each execution of the first optimization phase, excluding the first one.

We evaluate the impact in terms of time of the pruning approach in the PH-Louvain algorithm as before: we compare the algorithm with another test version in which we remove the pruning. In the Figure 23 there are the execution times of the first

optimization phase for each iteration of both two versions. As previous, we exclude from this graphic the first iteration.

We notice that the two algorithms perform generally in a similar way, with a significant difference only when the pruning reduce considerably the number of edge: the pruning algorithm is slower up to ~ 25 ms at iteration (in the graph dbpedia-link), caused by the extra time to compute the pruning information. Even if reducing the number of edges analyzed reduce the execution time, the gap between the two version is not directly proportional with respect to the number of edges pruned like in the previous version: this is due to the the branching problem for thread in the same warp.

During the Fill Map sub-phase, we insert in the map each pair node-community such that the nodes has almost one neighbour that change community in the previous iteration. To perform this operation, an edge $(source, destination)$ is assigned to each thread: than it check if the source node match the pruning requirement, it change the destination node with the destination communities and insert the value in the map. If an edge has a source node that doesn't match the criteria, its thread will return without doing nothing else. The problem is that the nodes are organized in warp of 32 threads assigned to a core of the Streaming Multiprocessor: the GPU execute a new warp on the same core only when all thread of the old warp finish the execution. If in a warp there is even just one thread that have to insert the pair in the map, all the other thread have to wait it. This reduce the gain in terms of times in the Fill-Map sub-phase.

We notice that in the no-pruning version the execution time tend to decrease even in the case in which the pruning doesn't remove a considerable part of the edges (like the wikipedia_link_jp dataset), so we analyze the sub-phases in order to see what change between each iteration: in the Figure 24 is illustrated the consuming time of each sub-phase with respect to the total time of each iteration. The most consuming sub-phase is the fill map: this sub-phase at least 50% of the time, reaching peek of even more than 75%. We notice that at each iteration, the times required decrease: therefore we focus our analysis on this behaviour of the map. We analyze the number of conflict that we have when we try to insert a new key in

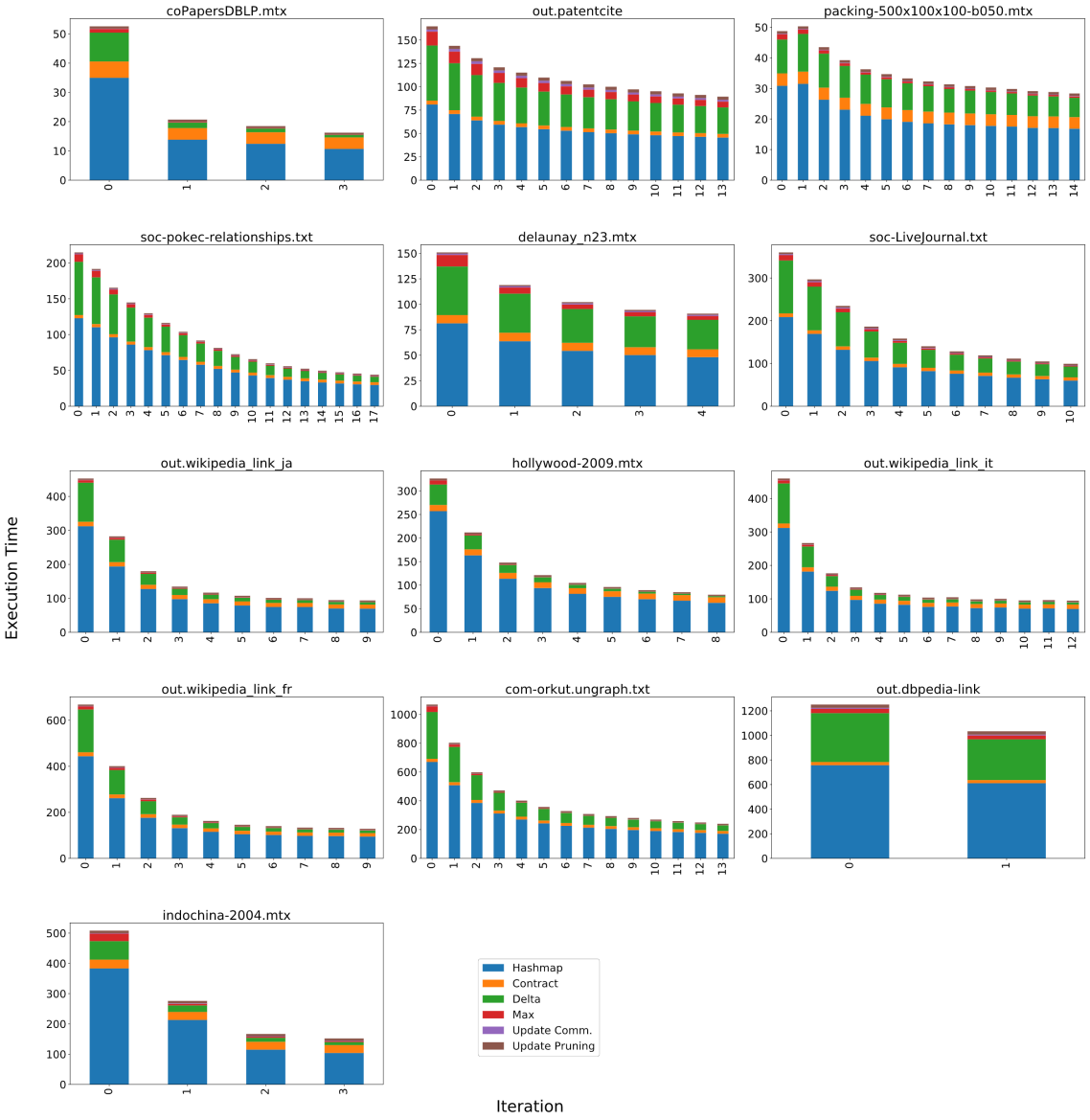


Figure 24

the map. We notice that, iteration by iteration, the number of conflicts in the map decrease: the reason is that, at each iteration, the number of different key inserted decrease. Indeed, at every iteration the number of communities decrease because similar nodes tends to converge to one community for groups: this reduce the number of possible different keys node-community that we have. Then, we analyze the number different keys inserted in the map at each iteration.

In the Figure 25, we have on the x-axis the number of different keys at that iteration with respect to the maximum possible number of keys (i.e. the number of edges, because we have the maximum when each node is in its self-community); on

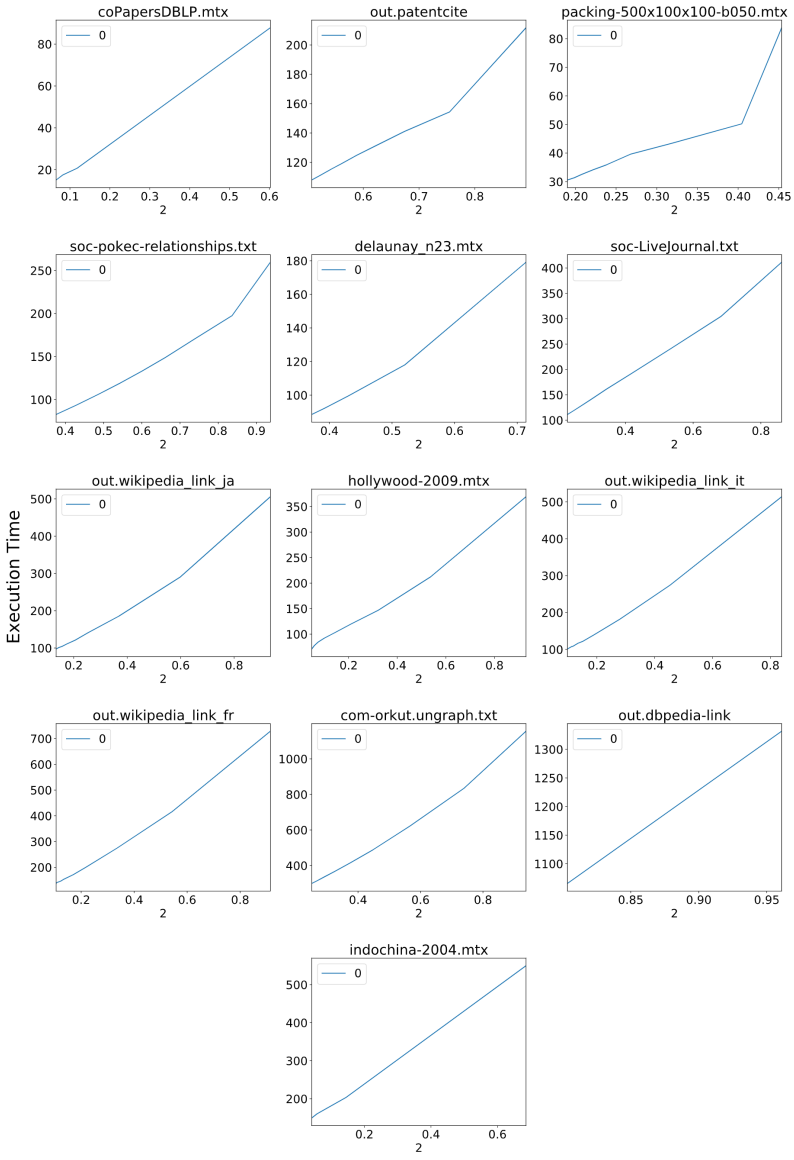


Figure 25: Execution time of the Hashmap version in relation to the number of different pair (node, community) inserted (expressed with respect to the maximum number of keys).

the y-axes we have the execution times. We notice that there is a direct correlation between this data and this two values, and this explains why the PH-Louvain algorithm performs better in the last iterations with respect to the first ones.

6.4 Algorithms comparison

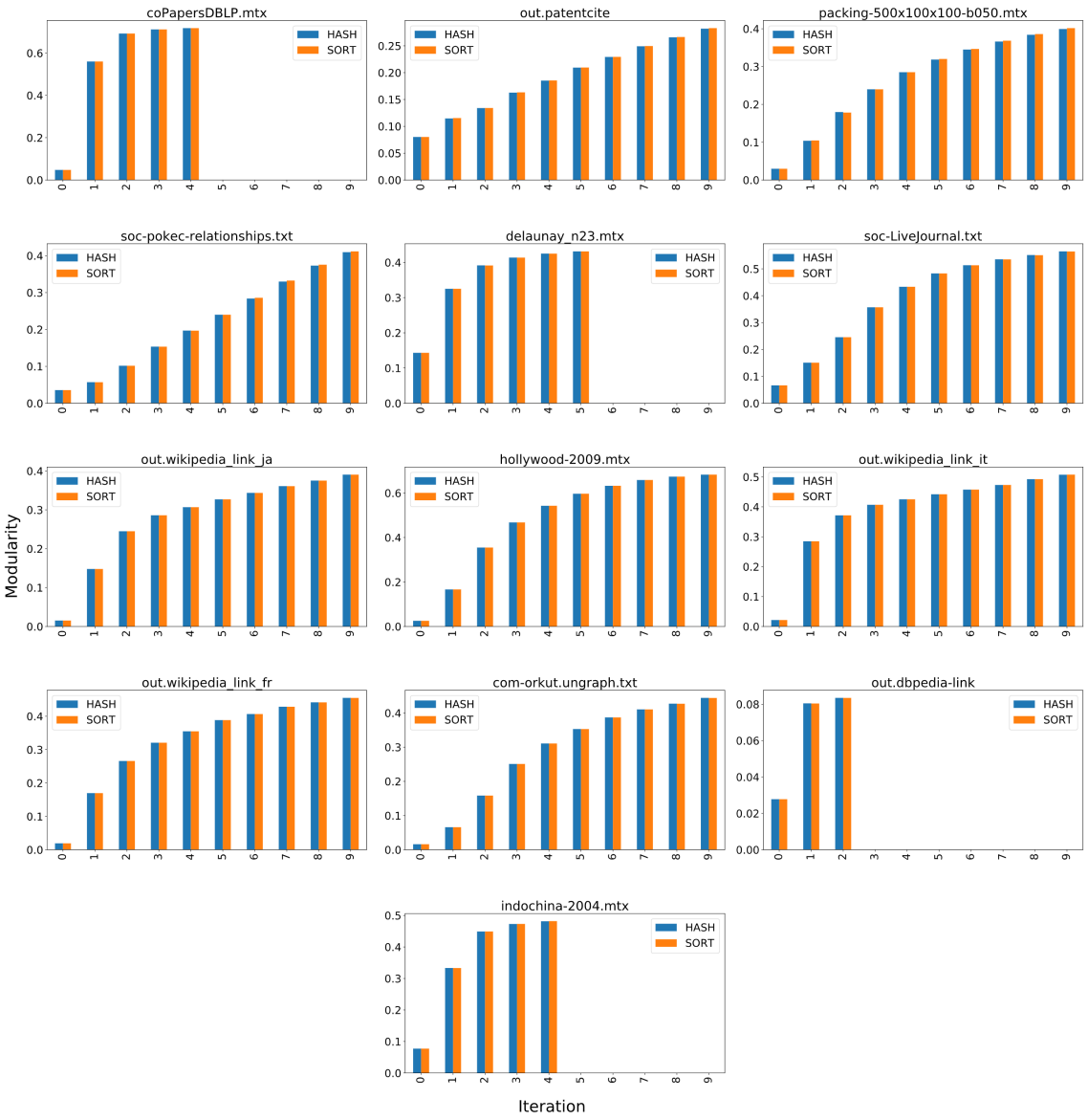


Figure 26: Modularity Progression in the first ten iteration of the optimization phase.

In this section, we focus our analysis on the comparison between the two algorithm in order to find advantages and disadvantages of each method. First of all, we notice from the Table 2 that the two algorithm obtain a very similar score of modularity. In the Figure 26, we expose the progression of the modularity Q in the first operation. As we can see, the modularity in the two algorithms grows in almost identical way: this is due to the minimum labelling heuristic (Chapter 4.3) that force the algorithms to converge to a similar result.

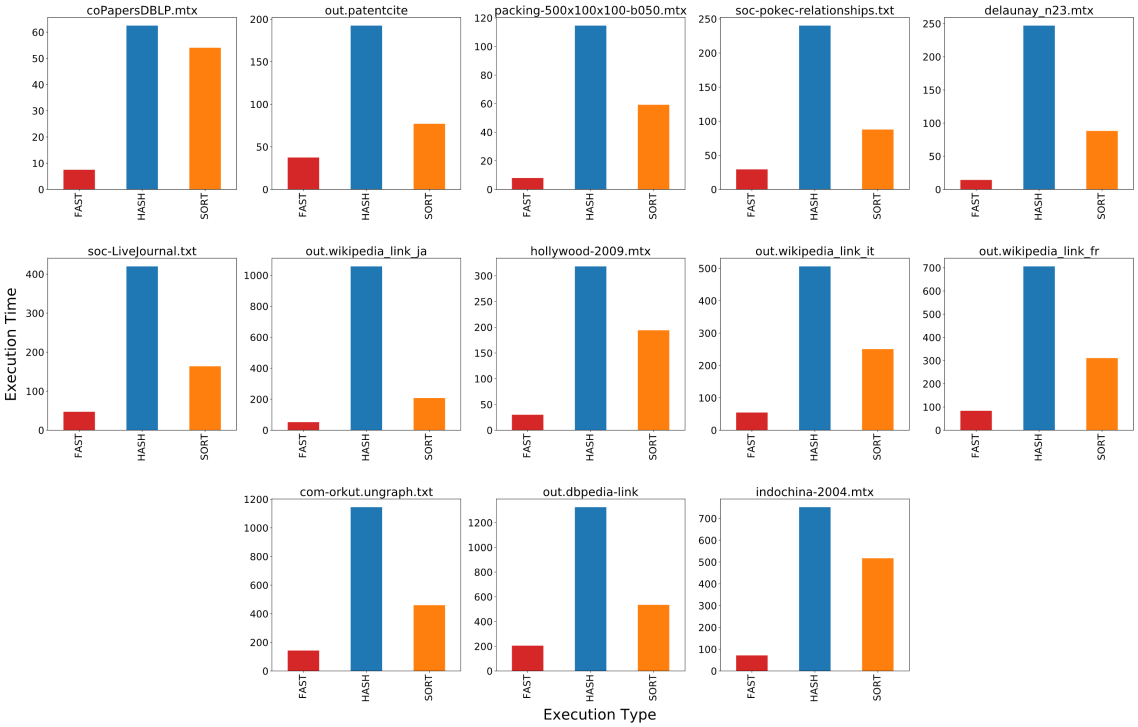


Figure 27: Comparison of the execution time of the first iteration of the first optimization phase between the two presented versions unoptimized of the algorithm and the optimized version.

To analyze the differences in terms of performance, we start analyzing the impact of the optimization of the first iteration of the optimization phase for both the algorithms. We are expecting an huge reduction in terms of times, considering that we remove the most consuming time phase from the Prune-Sort-Reduce routine, i.e. the sorting phase. As shown in the Figure 27, we obtain the expected results: we obtain a reduction in range between $\sim 51\%$ and $\sim 86\%$ respect to the PSR-Louvain and a reduction in range between $\sim 84\%$ and $\sim 95\%$ respect to the PH-Louvain. From the data we also see that in the second algorithm, this optimization introduce a little delay in the next iteration respect to the first one: this is caused by the deallocation of the support variable used in the fast approach (that are the same used in the PSR-Louvain) and the allocation of the hashmap and other support variables. But, this overhead is much smaller than the gain obtained by the optimization, therefore we consider it a valid technique to improve the performance.

From the Figure 27, we also note that the PH-Louvain is always slower with respect to the PSR-Louvain, and for this reason this technique has a greater impact on

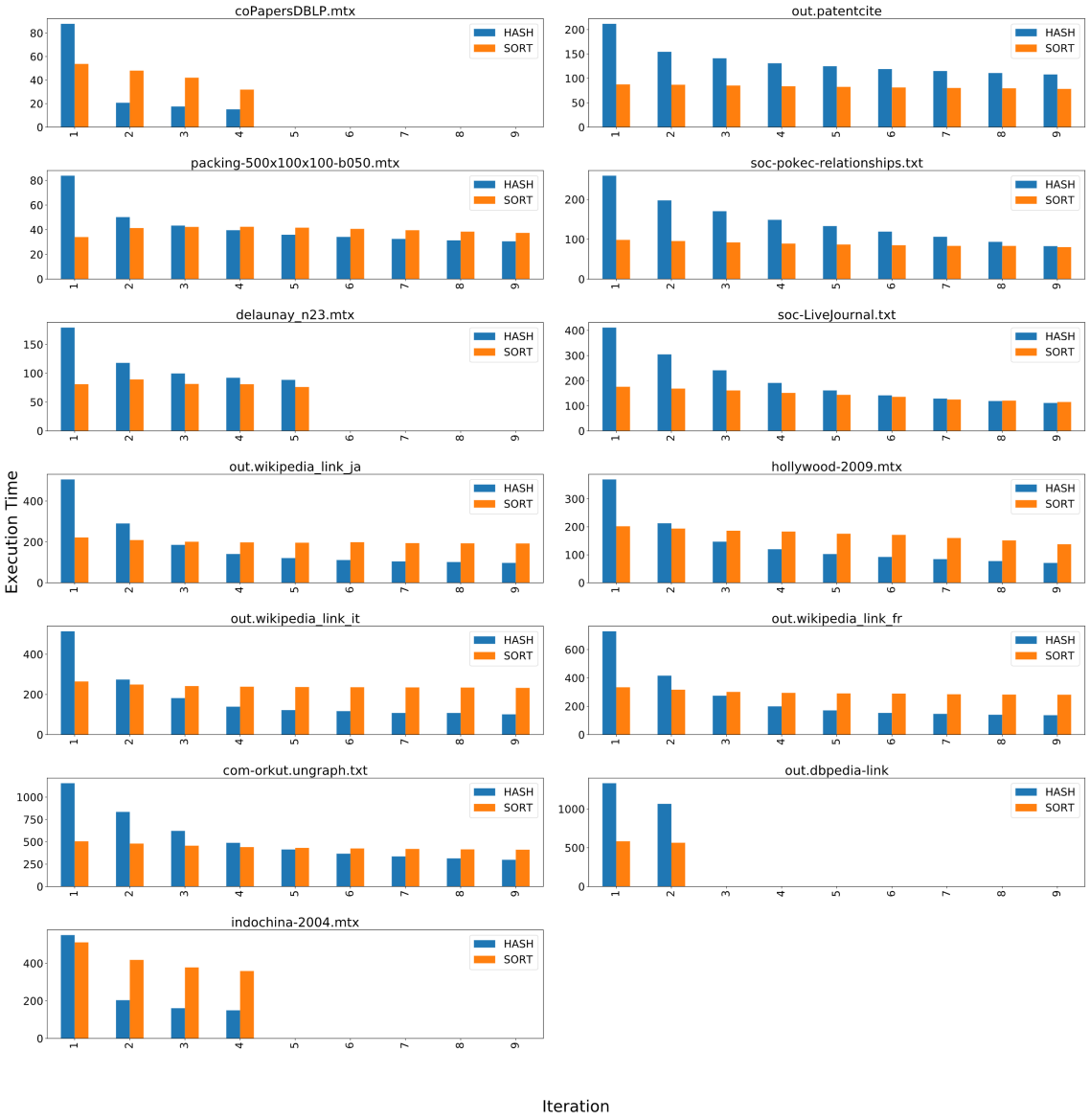
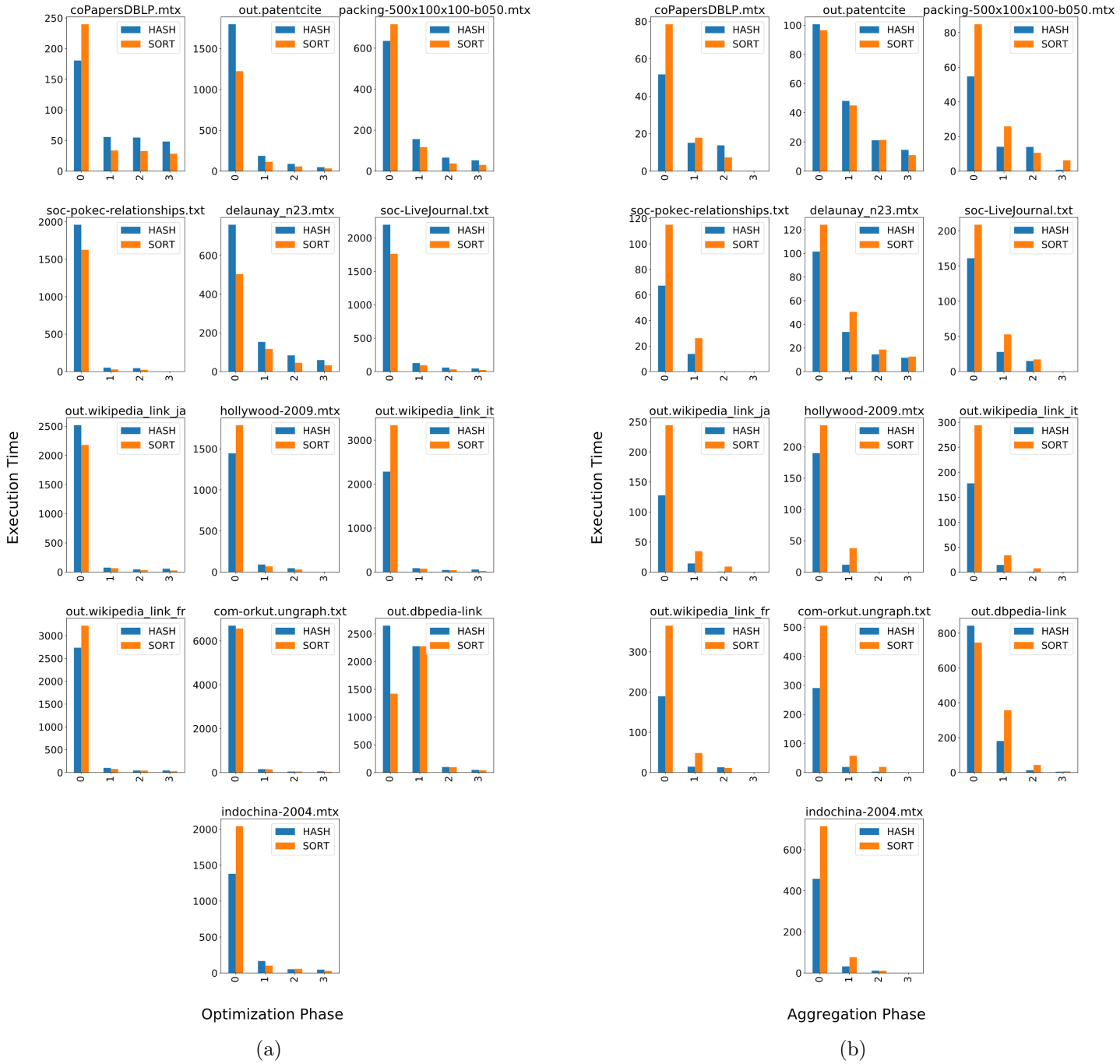


Figure 28: Execution time in the first ten iteration of the first optimization phase, excluding the first optimized iteration.

it. Considering this particular feature, we choose to go further in our analysis: we compare the iterations of the first optimization phase in terms of times of the two version of the algorithms, excluding the first iteration. In the Figure 28 are exposed the results. We notice that the second iteration of the PH-Louvain is always slower respect to the PSR-Louvain. Considering the observation presented in the previous chapter, we conclude in the first iterations the PH-Louvain algorithm suffer respect to the other due to the high number of different pair key-community that we insert in the map. On the other hand, if the number of different pair drops below a given

values, the hashmap performs better. Instead, the PSR-Louvain does not benefit from a similar behaviour and its performances are generally more stable (they are improved consistently only by the pruning). We analyze in a similar way also the other optimization phases (Figure 29a). We notice that, from the second optimization phase, generally the PSR-Louvain performs better in terms of total time (i.e. sum of the time of each iteration) than the PH-Louvain: this because the number of iteration generally decrease phase by phase and the first approach generally performs better than the second one in the first iterations of the phase. Even if the number of conflicts decrease (due to the fact that we allocate in a map with the same order of magnitude of the first phase in the following phases), also the time of the sorting sub-phase of the other algorithm decrease: to really see an improvement of the hashmap respect to the other approach we a similar convergence see in the first optimization phase. For these reasons the sorting version of the algorithm tend to perform better than the hashmap one from the second optimization phase.
We study also the difference between the algorithms in terms of aggregation phases: the results of our study are presented in Figure 29b. We notice that the execution times of the PH-Louvain tend to perform better with respect to the PSR-Louvain. Considering the observation about the performance of the hashmap that we make previously, its easy to motivate this performance results: the hashmap perform better respect to the sorting approach when the number of keys decrease consistently. In the aggregation phase, the number of the active communities is decreased compared to the starting situation. Besides, in this phase we insert in the map a key composed by the two communities id: this involves a further reduction of the number of keys inserted, because we tend to insert more often the same key in the map. Therefore the performance are related to the number of different active communities find at the end of the optimization phase: indeed the only two case in which the sort-reduce version perform better with respect to the hashmap version are those in which the number of communities is decreased slightly with respect to the number of edges (i.e. the maximum possible number of communities). All of these consideration are at the base of the design of an adaptive approach that we present in the next section: indeed, we can estimate if the PH-Louvain performs better then the



(a) (b)

Figure 29: (a): Execution times of the first four optimization phases; (b) Execution times of the first four aggregation phases.

other algorithm by the comparing of the number of different keys with respect to the number of the possible one, and select the right algorithm to use.

7 Adaptive Louvain Algorithm

In this chapter we finally present the adaptive algorithm: it uses an heuristic to switch from the sort-reduce version to the hashmap version when the number of different node-community keys falls below a certain threshold. This threshold is calculated in percentage with respect to the total number of possible keys, i.e. the number of edges. Next, we make a comparison between our adaptive algorithm and other two Louvain algorithms for the GPU already available in two important libraries: the first one is included in the NVIDIA's cuGraph library; the second one in the Gunrock library.

7.1 Algorithm

Our adaptive Louvain algorithm for the GPU combine the Prune-Sort-Reduce algorithm and the Hashmap algorithm in order to select the best behaviour basing the choice on the situation.
In the previous chapter we show that the Prune-Sort-Reduce algorithm performs better than the Hashmap in the first iteration of the optimization phase. We also show that, if the number of different key falls below a given threshold, the Hashmap version start to perform better than the Prune-Sort-Reduce version.
Based on these consideration, we design the optimization phase of our adaptive algorithm as following, bearing in mind that we keep iterate this phase until the difference in terms of modularity ΔQ between the iteration is greater than a given threshold $T_{\Delta Q}$:

1. At the beginning, we optimize the first iteration of the optimization phase using the optimized approach presented in Chapter 5.3.

2. Next, in the second iteration, we use the Prune-Sort-Reduce version of this phase, with an only addition respect to the standard version: we keep track of the number k^* of different tuples $(i, c_j, l_{i \to c_j})$that we have after the Reduce sub-phase, where n is the source node, c_j is the destination community and $l_{i \to c_j}$ is the sum of all the weights of edges that links the node i with a node

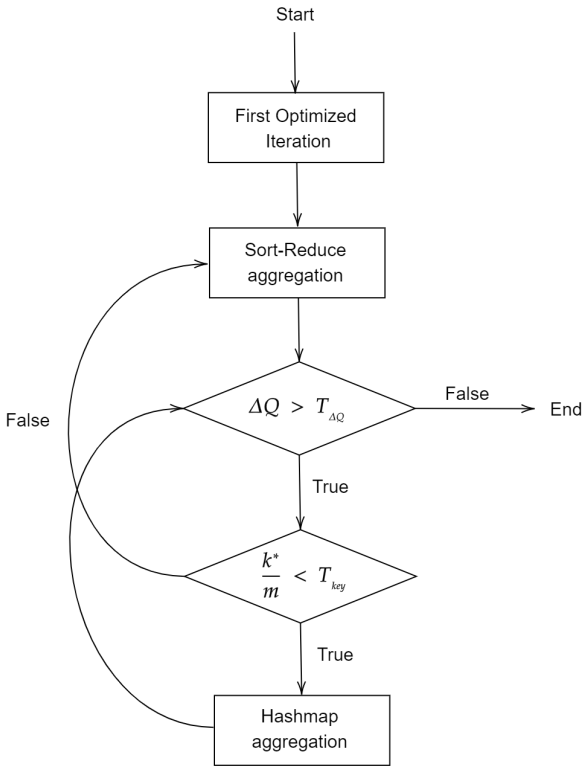


Figure 30: Schema of the Adaptive Optimization phase: before starting a new optimization routine, we check if the value $\frac{k^*}{m}$ falls below a given threshold: if it happen, we will use the hashmap based routine, otherwise we use the sort-reduce routine.

in the community c . This values is the number of different keys that that we would insert in the map if we used the other version of the optimization phase.

3. From the third iteration onwards, before anything else, we divide the value of k^* by the total number of edges m: the second values is equal to the maximum number of different keys that we can insert in the hashmap , i.e. the situation that each node is in each own community. If the value $\frac{k^*}{m}$ falls below a given threshold T_{key}, we execute the following iteration using the Hashmap approaches, otherwise we use the Prune-Sort-Reduce one.

The Figure 30 shows a schematic diagram of the optimization phase logic. Now all it takes is to find the right values of T_{key} that allow us to switch between the hashmap-based version when performs better. From the data presented in the next chapter, we observe that the two version take about the same time to perform an iteration of optimization when $\frac{k^*}{m}$ is between 0.3 and 0.4. When $\frac{k^*}{m} > 0.4$, the sort-

reduce approach performs better; when $\frac{k^*}{m} < 0.3$, the hashmap approach performs better. So we fixed the value of T_{key} equals to 0.3.

As regards the aggregation phase, we use the hashmap based approach in the adaptive algorithm, because, still from the observations presented in the previous chapter, we notice that this version tends to perform better with respect to the sort-version thanks to the reduction of the possible number of keys that we insert caused by the reduction of possible communities.

7.2 Analysis

Execution Times (in milliseconds) and Modularity			
Graph	Adaptive	PSR-Louvain	PH-Louvain
coPapersDBLP	369.14 0.8543	419.59 0.8544	412.79 0.8544
patentCite	1 660.79 0.7927	1 796.88 0.7911	2 555.14 0.7878
packing-500x100x100-b050	993.35 0.9403	1 045.25 0.9434	1 090.03 0.9416
soc-pokec-relationship	1 636.53 0.6935	1 843.95 0.6934	2 186.81 0.6852
delaunay_n23	987.63 0.9856	1 020.23 0.9856	1 319.22 0.9857
soc-LiveJournal1	2 078.29 0.7493	2 187.45 0.7491	2 677.51 0.7482
wikipedia_link_ja	1 919.34 0.5691	2 654.88 0.5690	2 305.11 0.5724
hollywood-2009	1 685.98 0.7542	2 092.09 0.7542	1 758.27 0.7542
wikipedia_link_it	2 417.01 0.7190	3 875.04 0.7190	2 732.99 0.7196
wikipedia_link_fr	2 741.07 0.6834	3 910.95 0.6834	3 273.91 0.6871
com-orkut	6 282.37 0.6616	7 566.90 0.6613	7 484.10 0.6629
wikipedia_en(dbpedia)	4 655.72 0.6612	5 287.65 0.6612	6 464.03 0.6618
indochina-2004	2 277.44 0.9632	2 899.50 0.9632	2 303.52 0.9632

Table 4: Adaptive vs PSR-Louvain vs PH-Louvain

In this section we analyze the results obtained by the Adaptive algorithm: we make a comparison between this version and the two other version presented previously in this thesis; besides we make a comparison with two interesting library: Nvidia's cuGraph and Gunrock. To make our test we use the same 13 datasets presented in Chapter 6.1 and the same Ubuntu 18.04.4 LTS machine used in the test presented previously. We remind that the machine is equipped with a 2.10GHz Intel(R) Xeon(R) Silver 4110 CPU, a TITAN Xp GPU with 12 GB of memory and CUDA 10.2. We start comparing and analyzing our adaptive respect to PSR-Louvain and the PH-Louvain presented in Chapter 5. We present the results in terms of time and in terms of modularity in the Table 4. We use a threshold $T_{\Delta Q}$ equals to 0.01. As we can see, we have an improvement in terms of times without any loss in terms of modularity: this is thanks to correctness of the params T_{key} that allow us to switch between the two approaches when it's needed. We study in detail the behaviour of the algorithm: Figure 31 shows the execution time of the first ten iteration of the first optimization phase for the PH-Louvain (in blue), the PSR-Louvain version (in Orange) and the Adaptive version (in green). The red dotted vertical line highlight when the Adaptive algorithm switch behaviour from the sort-reduce one to the hashmap one. When the algorithm switch from the first approach to the second one, the algorithm doesn't choose the first one until the next phase. We notice that in the dataset in which the sort-reduce version performs better (for example in the dataset wikipedia_en (dbpedia)) the algorithm use only this version and doesn't perform any change. In the other cases, we notice that the first iteration with the hashmap aggregator of the Adaptive algorithm show a little delay with respect to the same iteration of the PH algorithm: this is due to the deallocation of the support variable used in the sort-reduce approach and the allocation of the hashmap variables, as in the second iteration of the PH-Louvain, after the first optimized phase (Chapter 6.4). Besides, we investigate the behaviour of the algorithm also in the other optimization phase, in order to explain why the Adaptive version performs better than the other two approaches also in the dataset in which the PH-Louvain performs better in the first iteration phase, as indochina-2004. We notice that the algorithm uses the sort-reduce approach in the phases subsequent to after the first

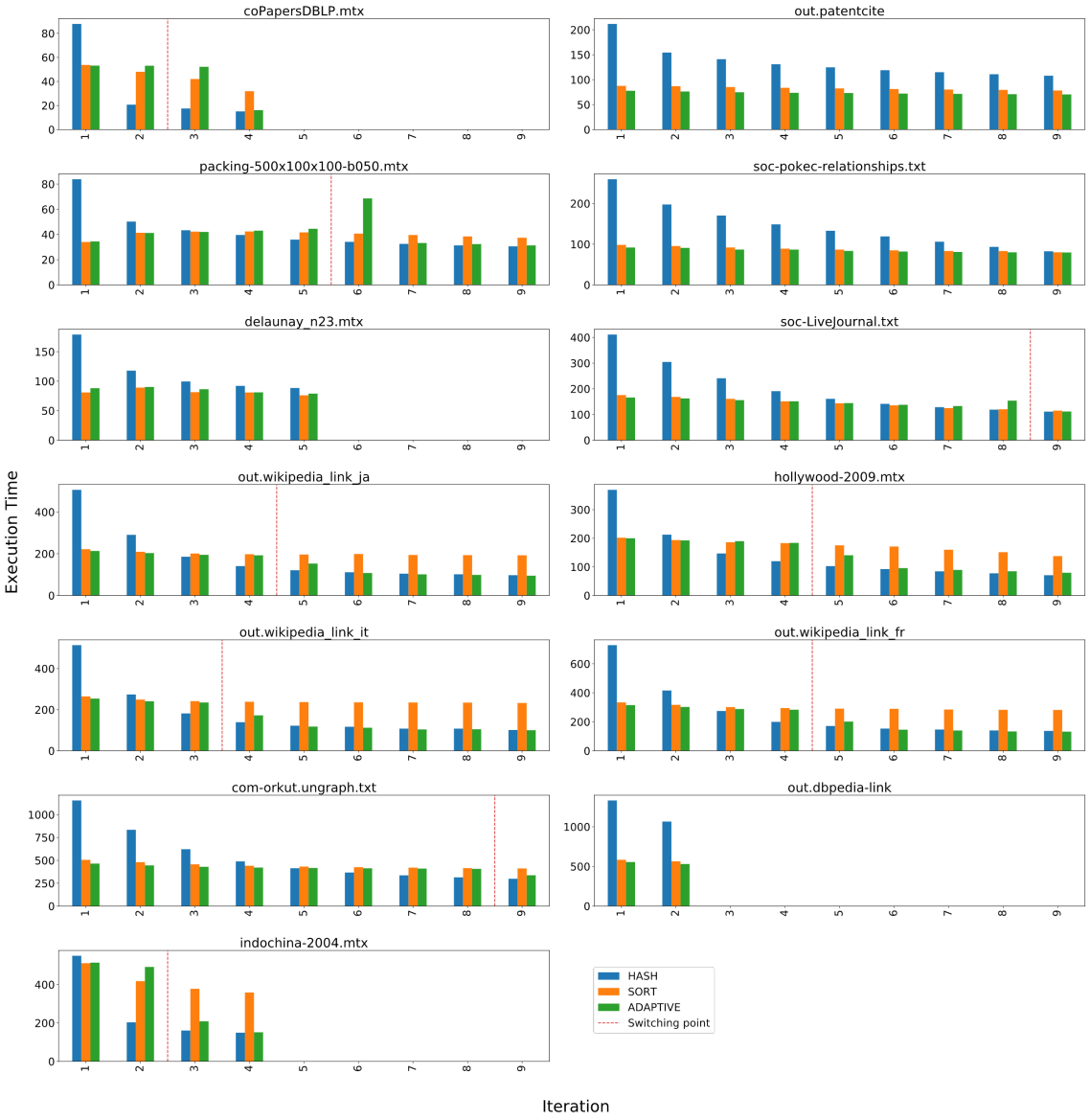


Figure 31: Execution time in the first ten iteration of the first optimization phase, excluding the first optimized iteration. The dotted red line indicate that, from the next iteration, the Adaptive algorithm uses a Hashmap based approach. Before the line the algorithm uses a Sort-Reduce approach.

for the most part of the iterations: this caused the increment of the performances because, as we present in the Chapter 6.4, the sort-reduce approach tend to perform better in the first iterations of the phases, and these phases tends to execute less iteration.

Now we present a comparison between our algorithm and other two version of the Louvain Algorithm for the GPU: the first one is part of the Nvidia's RAPIDS cuGraph library, a collection of GPU accelerated graph algorithms, actually still under

development but aviable in with a open-source license on GitHub; the second one is part of the Gunrock library, a CUDA library for graph-processing designed specifically for the GPU released in June 2019. The Gunrock team is one of the main contributors of the cuGraph library and some modules of the first library is integrated in the second. Both the algorithm doesn't implement any pruning approach or any adaptive approach.

In the optimization phase, the first algorithm use an hashmap for each node to aggregate the tuple (node, community, weight): each map is assigned to a unique vertex and the algorithm insert the weight using only the community as key. If there is already an entry in the map, the map automatically sum up the values. After this, the algorithm calculate the delta for each entry of the map and than select the maximum. To implement the hashmaps, they use two vector of lenght m, where m is the number of nodes, and assign to each node a private part of them in which insert its keys and values. They assign to each vector a space of size equals to the size of its neighbourhood and use an open-addressing procedure for the conflict, checking the subsequent free position. After this, they use a procedure similar to the Select Max sub-phase of the Prune-Sort-Reduce algorithm: they use the method `thrust::reduce_by_key` to get the best values using the two vector as values and as a keys a vector that keep track to which source node is assigned each position in the hashmap vectors. Besides, this algorithm propose a different techniques to avoid simultaneous swap: instead of forcing the node in single communities to select a community with a greater id like us, they permit to the nodes to select a community with a greater id only in the even iteration and to select a community with a lower id only in the odd iteration.
To contract the graph, this algorithm use a procedure similar to the one proposed in the Prune-Sort-Reduce algorithm: first they renumber the communities, then they create a vector of (community source, community destination, weights) from the edges vector replacing each node with the relative communities, and finally they sort and reduce this vector.

The Gunrock algorithm use a logic similar to the our Prune-Sort-Reduce version to aggregate the nodes:they copy the edges changing the destination node with the

corresponding community, than they sort all key-value pairs, and they use segmented reduce to accumulate the values in the continuous segments. Finally the algorithm use a reduction by keys to select the maximum value. To aggregate the graph, they use a procedure similar to cuGraph and our Prune-Sort-Reduce version. Our study to this algorithm find an error in the stopping criteria logic of the optimization phase: this algorithm, to calculate the difference ΔQ between two configuration, sum all the various $\Delta Q_{i \to c_z^*}$ for each node i that change its community to c_z^* obtained during the optimization phase to the previous modularity value. This values is incorrect because, changing all the communities simultaneously, the various l and k_c values of the equation 22 are referred to a situation that isn't the actual any more. This causes a wrong calculation of the temporary modularity score, that can also exceed its upper bound of one during this phase. This causes also a reduction in the execution time because this algorithm doesn't recalculate the modularity after each iteration. Nevertheless, our tests on this library highlight that the algorithm doesn't iterate the optimization phase more than 10 times: this early stop criteria, combined with a recalculation of the final score after the optimization, allows the algorithm to find a valid configuration.

To setup our test in order to have comparable results between these three versions, we set the threshold $T_{\Delta Q}$ equals to 0.01 for the Adaptive and cuGraph version. To set this value we change the source code of the library because this value is fixed in the actual release. Using this parameters, the execution time and the final modularity are comparable with the Gunrock version. We present the results in the Table 5. The first thing that we notice is that our algorithm is better optimized in terms of memory: on our machine we can analyze graphs up to $\sim$ 175000000 edges, instead our competitor go out of memory when the graph has $\sim$ 90000000. One of the reasons that allow us to reach graph of bigger size is the memory optimization of the our optimization phase: therefore, we divide the edges in some buckets of fixed size, we execute the phase, store the results and start the execution on another bucket; when all buckets are processed we update the communities. With this technique we reduce the amount of memory used to store the data in that phase from about $2m$, where m is the number of edges of the graph, to $2 * s + n$, where s is the bucket size

Execution Times (in milliseconds) and Modularity			
Graph	Adaptive	cuGraph	Gunrock
coPapersDBLP	369.14 0.8543	589.27 0.8541	561.15 0.8416
patentCite	1 660.79 0.7927	1 175.62 0.7936	1 615.13 0.7806
packing-500x100x100-b050	993.35 0.9403	885.77 0.9437	734.37 0.9372
soc-pokec-relationship	1 636.53 0.6935	1 546.91 0.7109	1 019.44 0.6935
delaunay_n23	987.63 0.9856	1 041.23 0.9877	1 608.63 0.9860
soc-LiveJournal1	2 078.29 0.7493	2 474.60 0.7504	2 226.35 0.7504
wikipedia_link_ja	1 919.34 0.5691	2 744.37 0.5825	2 283.09 0.5647
hollywood-2009	1 685.98 0.7542	2 210.50 0.7511	2 028.64 0.7468
wikipedia_link_it	2 417.01 0.7190	4 184.10 0.7326	2 653.13 0.7112
wikipedia_link_fr	2 741.07 0.6834	4 017.74 0.6848	3 392.85 0.6759
com-orkut	6 282.37 0.6616	- -	- -
wikipedia_en(dbpedia)	4 655.72 0.6612	- -	- -
indochina-2004	2 277.44 0.9632	- -	- -

Table 5: Adaptive vs cuGraph vs Gunrock

and n is the number of nodes (Chapter 5.4). Besides, our test highlight that we use fewer memory to store the graph in the device memory.

In terms of modularity we notice that our algorithm is on pair with the cuGraph version that, for the reason presented previously, is more accurate than the Gunrock algorithm. In terms of performance we notice that our algorithm perform better to the bigger graphs. Analyzing the cuGraph optimization phase, we notice that this version tend to perform more iteration with respect to the Adaptive version: this behaviour is probably caused by the fact that they allow each node to select a community with a greater id or a smaller id intermittently. We also notice that their optimization iteration tends to go faster iteration by iteration, acting as our

Hashmap algorithm (Chapter 6.3.2); furthermore, the optimization phase are generally slightly faster with respect to our hashmap approaches at the same iteration, and so this algorithm is quicker than ours when both performs the same number of iterations. It's not clear if the improvement is made by internal optimization of the operation or thanks to the fact that they uses an hashmap for each node instead of a global one: the analysis of this is left as future work. Finally we notice that the our algorithm contract the graph quickly by the using the hashmap approach instead of the sorting one (Chapter 6.4).

Analyzing the Gunrock algorithm, we notice this algorithm does not calculating the modularity after each step and it hav an early stop at the tenth iteration: this causes a gain in term of time with respect of our algorithm if we execute more than ten iteration without swapping behaviour to hash-mode (or when the gain that we have using map is not high) , thanks to the time saved from the modularity calculation. On the other hand, they doesn't stop the iteration when they should: this can lead to perform extra useless iteration of the optimization phase. In the wrong case, this behaviour can also to a decrease of global modularity between two iteration and therefore to doesn't find the local maximum. Surprisingly, this algorithm find however a good modularity score. We left as future work the analysis of this behaviour, and if we can avoid to perform the modularity calculation at each iteration to reduce the execution time without significant loss in modularity.

Concluding, our adaptive algorithm performs very well even compared to a some Louvain algorithm presented in some library related with Nvidia: we even occupy fewer memory and we perform better on the largest graphs.
The pruning approach helps the algorithm to reduce time, but it highly depends on the graphs. This approach can be very useful in multi-gpu algorithms, where one of the bottleneck is the data transfer between the global memory to the GPU, even using the an hashmap to aggregate, that our tests show that have a smaller improvement with respect to the sort-reduce aggregation technique. This technique can be useful also to compute the modularity on graph that doesn't fit in the device memory. We left the design and the analysis of that algorithm as a future work.
The adaptive behaviour is very effective and allow us to reduce the computation

time of the first inefficient operation with the hashmap. We left as future work the analysis of the performance of using a private hashmap for each node instead a global one, and the impact using this behaviour on a adaptive approach.

8 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. "Fast algorithm for detecting community structure in networks". In: *Physical Review E* 69.6 (June 2004). ISSN: 1550-2376. DOI: 10.1103/physreve.69.066133. URL: http://dx.doi.org/10.1103/PhysRevE.69.066133.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663.

[18] Dan A Alcantara et al. "Building an efficient hash table on the GPU". In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 39–53.

[19] 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.

[20] Jérôme Kunegis. "Konect: the koblenz network collection". In: *Proceedings of the 22nd International Conference on World Wide Web*. 2013, pp. 1343–1350.

[21] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] *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.

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