# Scalable Subgraph Counting using MapReduce

Ahmad Naser eddin, Pedro Ribeiro
CRACS & INESC-TEC
DCC-FCUP, Universidade do Porto, Portugal
ahmadnasereddin.ie@gmail.com, pribeiro@dcc.fc.up.pt

## ABSTRACT

Networks are powerful in representing a wide variety of systems in many fields of study. Networks are composed of smaller substructures (subgraphs) that characterize them and give important information related to their topology and functionality. Therefore, discovering and counting these subgraph patterns is very important towards mining the features of networks. Algorithmically, subgraph counting in a network is a computationally hard problem and the needed execution time grows exponentially as the size of the subgraph or the network increases.

The main goal of this paper is to contribute towards subgraph search, by providing an accessible and scalable parallel methodology for counting subgraphs. For that we present a dynamic iterative MapReduce strategy to parallelize algorithms that induce an unbalanced search tree, and apply it in the subgraph counting realm. At the core of our methods lies the g-trie, a state-of-the-art data structure that was created precisely for this task. Our strategy employs an adaptive time threshold and an efficient work-sharing mechanism to dynamically do load balancing between the workers.

We evaluate our implementations using Spark on a large set of representative complex networks from different fields. The results obtained are very promising and we achieved a consistent and almost linear speedup up to 32 cores, with an average efficiency close to 80%. To the best of our knowledge this is the fastest and most scalable method for subgraph counting within the MapReduce programming model.

## CCS Concepts

•**Mathematics of computing → Graph algorithms;**
•**Theory of computation → MapReduce algorithms;**

## Keywords

Subgraph Search, Motif Discovery, Complex Networks, Graph Mining, G-Tries, Parallel Algorithms, MapReduce

*SAC 2017,April 03-07, 2017, Marrakech, Morocco*
Copyright 2017 ACM 978-1-4503-4486-9/17/04...$15.00
http://dx.doi.org/10.1145/3019612.3019744

## 1. INTRODUCTION

In recent years network science has emerged as an important multidisciplinary field, with applications in areas such as computer science, physics, biology or engineering. Although its roots are in the older field of graph theory, the analysis of networks has been recently receiving increasing attention due mainly to two important factors [4].

The first contributing factor is the availability of network maps, due to technological advances that have provided an enormous amount of data which can be represented by networks. The second contributing factor is the realization that complex networks from different areas share non-trivial common topological features. This universality of characteristics serves as a guiding principle for network analysis and gives a wide applicability to any discoveries.

In order to extract information from networks, practitioners have a wide range of measurements available [8]. Some of them describe properties at the node level (such as its degree) while others describe global metrics (such as the average distance between nodes). One other way of analyzing a network is to use an intermediate approach, looking at small topological patterns of interconnections, bigger than a single node but smaller than an entire network, and trying to understand what is their role in the global behavior of the network. These small substructures are subgraphs and they can be seen as basic building blocks of complex networks, capable of uncovering their design principles [15].

One crucial related task is the computation of subgraph frequencies. For instance, the concept of *network motifs*[15] points towards subgraphs that appear in significantly higher numbers than what one would expect. To discover motifs we therefore need the ability to count subgraphs. Another example are *graphlet degree distributions* [18], which at its computational core lies around discovering occurrences of subgraphs.

The task of computing the frequency of a given set of subgraphs (also known as computing a *subgraph census*) is a challenge, since it is a *computationally hard* problem. In fact, just knowing if a subgraph appears at all in another larger graph is an NP-complete problem [7], and finding the exact number of times it appears is an even harder task. Given this, the needed execution time grows exponentially as we increase the size of the network or the size of the subgraphs being searched.

One of way of improving the performance of the associated algorithms is to resort to parallelism, which has the capability to really scale up the computation. In this paper we aim precisely towards parallel subgraph search and we use the MapReduce programming model [6]. An important advantage of MapReduce is its availability on cloud computing websites. For instance the user could rent computation time in a cluster in any cloud computing provider (e.g. amazon web services) and run the algorithm using as many processors as needed. So the user does not need to have his own powerful machines to run large graphs. By providing subgraph counting algorithms in a MapReduce framework we could therefore really make them available to a wider and more general audience of practitioners in different fields.

The main contributions of the work described in this paper can be summarized as follows:

- A general iterative MapReduce parallel strategy for unbalanced *"tree-like"* computations. It employs a dynamic threshold that changes during the execution and adapts the time between the iterations depending on the actual computation being made. It also redistributes work after each iteration, effectively providing dynamic load balancing.
- A Java implementation for the Spark framework that uses the developed strategy and applies it for subgraph counting using g-tries. It includes a compact representation of the work state, allowing a very efficient work sharing mechanism.
- A thorough experimental analysis of our implementation on a large set of representative complex networks, demonstrating its general applicability and showcasing its scalability.

The rest of the paper is organized as follows. Section 2 introduces the problem being solved. Section 3 describes the base sequential algorithm, while Section 4 details our parallel approach. Section 5 gives our experimental results and finally Section 6 concludes the paper.

## 2. SUBGRAPH COUNTING

In this section we introduce a common graph terminology to be used throughout the paper, we formally define the problem we are tackling and we talk about past related work.

### 2.1 Graph Terminology

A graph $G$ is composed of a collection of vertices $V(G)$ and a set of edges $E(G)$. The *size* of the graph is the number of vertices it has, and it is written as $|V(G)|$. The term *k-graph* refers to a graph of size $k$. Edges are composed of pairs of vertices $(u, v)$. The *degree* of a vertex is the number of edges it has. In the case of directed graphs we can distinguish between the *indegree* (incoming edges) and the *outdegree* (number of outgoing edges). The *neighbourhood* of a *vertex* $u$ is the set of vertices that share an edge with $u$. If the graph has no self-loops or multiple edges connecting the same pair of vertices, then it is considered a *simple graph.* In this paper we assume that we are only dealing with simple graphs.

Vertices are distinguished by assigning them *labels* from 0 to $|V(G)|-1$, thus the comparison $u < v$ refers to a comparison between their labels and in this case it means that the vertex $u$ has a lower label than $v$. These labels are used as part of the g-tries symmetry breaking conditions, which allow the algorithms to only count each subgraph occurrence once.

Some graphs contain other graphs. The contained graph is called a *subgraph.* Formally, a graph $H$ is a *subgraph* of a graph $G$ if $V(H) \subset V(G)$ and $E(H) \subset E(G)$. This subgraph $H$ is called *induced* if $\forall u, v \in V(G), (u, v) \in E(H)$ if and only if $(u, v) \in E(G)$, and if the graph $G$ has a set of nodes that induce $H$ then this set is called an *occurrence* or a *match.* Distinct matches must have at least one different vertex. The number of occurrences of $H$ in $G$ is called its *frequency.* Figure 1 shows an example of a graph $G$, a subgraph $H$ and its four occurrences.

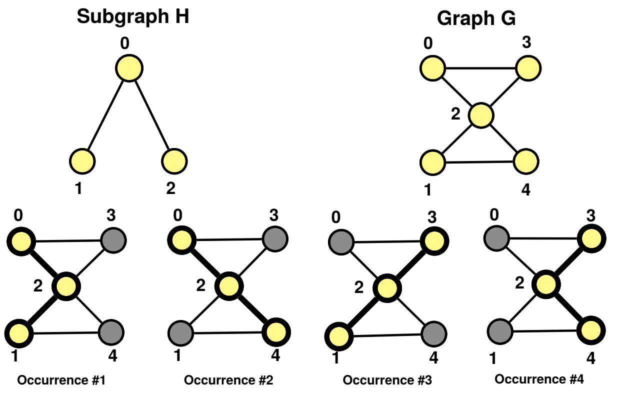

**Figure 1:** Induced occurrences of a subgraph $H$ in a larger graph $G$.

Two graphs $G_1$ and $G_2$ are *isomorphic* ($G1 \sim G2$) if there is a one to one mapping between their vertices and there is an edge in $G_1$ if and only if there is an edge between the corresponding vertices in $G_2$. This problem (*isomorphism*) is computationally *hard* and it is neither known to be solvable in polynomial time nor NP-complete [14]. Another similar but different problem is *subgraph isomorphism*, in which given two graphs $G_1$ and $G_2$ we need to determine if $G_1$ contains a subgraph which is isomorphic to $G_2$. This problem is known to be NP-complete [7].

### 2.2 Subgraph Census Problem

As the last section described, just knowing wether a graph appears as a subgraph of another larger graph is already an NP-complete problem. The main computational problem that we are trying to solve in this paper is an even more general version of this problem, that is, to actually compute the number of occurrences of each subgraph type. Our goal is precisely to improve the efficiency and scalability of algorithms for this task and we now define more formally the problem we are tackling.

DEFINITION 1 (**Subgraph Census Problem**). *Given a graph $G$ and a subgraph of size $k$, determine the exact frequencies of all induced occurrences of all possible $k$-subgraph types in $G$.*

In some cases we may instead be interested in a smaller set of subgraphs than the entire set of size $k$. Note that the number of different subgraph types grows exponentially as $k$ grows and that this number is different between the undirected and directed case. Figure 2 exemplifies this by displaying all possible subgraph types of size 3.

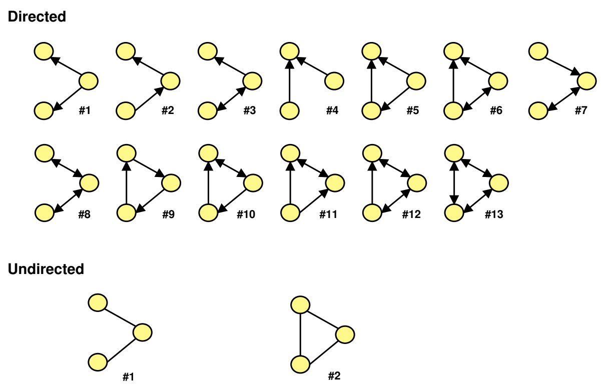

**Figure 2:** All possible directed and undirected 3-subgraphs.

## 2.3 Related Work

There is a significant amount of past work on sequential subgraph census algorithms, which can be divided into three main conceptual approaches.

**Network-centric** methods search for all subgraphs of size $k$ in the target network and then apply isomorphism to determine the type of each subgraph in the occurrences. Example of this class of algorithms are ESU [25], Quatexelero [11] or FaSE [17].

**Subgraph-centric** methods search only for one individual subgraph type at a time. An example algorithm of this type is Grochow and Kellis [10].

**Set-centric** methods search for a customized set of subgraphs. They are conceptually in the middle between the previous two, because they do not search for only one subgraph at a time, and also not necessarily for all subgraphs of a specific size. This approach was introduced in 2010 with the usage of the g-trie data structure providing an efficient way of representing general sets of subgraphs [20].

In this work we opted to use the set-centric approach as the sequential baseline, given its flexibility on what subgraphs should be counted. Furthermore, to the best of our knowledge, g-tries are currently the state-of-the-art in what concerns subgraph counting for a general scenario (no specific subgraph types or sizes) [21], providing a fast and powerful methodology for subgraph search, including the possibility of trading accuracy for even more speed by using sampling techniques which lead to approximate results [19].

Regarding past parallel approaches, the methods we develop here are different and novel in fundamental ways. In what concerns g-tries, there already exists a distributed memory approach using MPI [22] and a shared memory approach for multicores using pthreads [2]. Both of these approaches rely on a form of work stealing for providing load balancing. By contrast, our work is geared towards a MapReduce programming model, which uses a different underlying abstraction and therefore requires different parallel strategies.

In what concerns MapReduce approaches for subgraph census, there is no previous work on set-centric methodologies. We recently became aware of MRSUB [23], which uses a network-centric approach with an edge-based enumeration, but its base sequential algorithm is substantially slower than g-tries. MRSUB also uses a static load balancing scheme and a single MapReduce iteration, while in our case we use a dynamic load balancing scheme that adapts to the computation during execution. Another MapReduce network-centric approach is given in [24], in which they use an iterative approach. However, they use as a sequential basis the ESU algorithm (two orders of magnitude slower than g-tries) and with 56 cores their reported speedup was on average smaller than 5x (with the exception of a specific network, where they reported a speedup of 37.81x). Regarding subgraph-centric Map-Reduce approaches, there are options such as [12] or [1]. They differ fundamentally from our work because they are trying to parallelize the enumeration of a single subgraph. This conceptual methodology implies that we need to search for all subgraphs individually when doing a subgraph census, not reusing any kind of computation between enumerations, which makes this approach less desirable when the number of different subgraph types is high.

We should also mention that there are parallel approaches for less general versions of subgraph search when we are only interested in specific types of sugraphs. An examples of this is Sahad [26], a subgraph-centric MapReduce approach for tree subgraphs.

## 3. SEQUENTIAL G-TRIES ALGORITHM

In this section we describe our base sequential algorithm, centered around the very efficient g-trie data structure [21]. Its core advantage lies in the fact that it identifies common substructures between different subgraphs and this is used for searching at the same time for any given set of subgraphs. We will exemplify it with undirected subgraphs, but the ideas are easily extendable to the directed case.

### 3.1 Structure of a G-Trie

G-tries are similar in spirit to the idea of prefix trees [9]. However, instead of storing a collection of strings and identifying common prefixes, it stores a collection of subgraphs and identifies common subtopologies. Like the prefix tree, it is a multiway tree, but instead of adding one character in each child node, it adds one vertex and its correspondent connections to ancestor vertices. A path from the root to a leaf defines a subgraph and descendants from any g-trie branch have some commmon subtopology.

Figure 3 shows an example of a g-trie containing 6 undirected 4-subgraphs. It also illustrates how g-tries are augmented with symmetry breaking conditions that induce a unique order in which vertices can match a subgraph. This plays a big role on g-tries efficiency and makes it possible to count each occurrence only once. For example, subgraph type #6, a clique of size 4, has the set of conditions $A < B, B < C, C < D$. Without these, any permutation of the nodes of an occurrence would also be a match, since this subgraph is completely symmetric and all nodes are topologically equivalent.

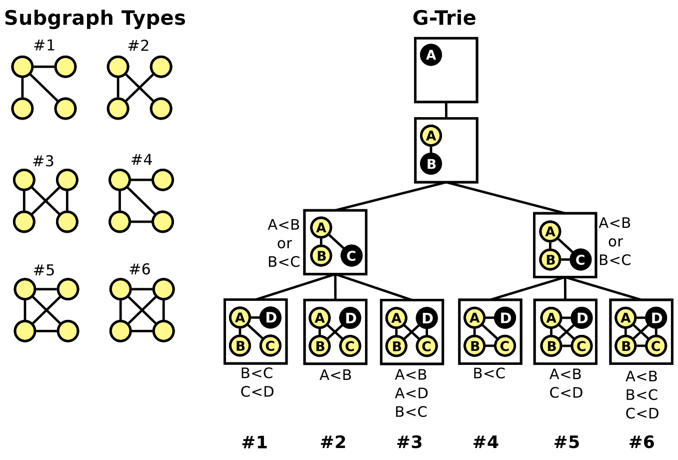

**Figure 3:** A g-trie representing a set of 6 undirected subgraphs of size 4. Each g-trie node adds a new vertex (in black) to the already existing ones. Clauses of the form $X < Y$ are symmetry breaking conditions. Adapted from [2].

## 3.2 Using G-Tries to Count Subgraphs

Algorithm 1 details how we can use g-tries to count subgraphs. The core idea is that we will be searching for a set of network nodes ($V_{used}$) that match to a certain g-trie path ($T$). This search is heavily constrained (see function `matchingVertices()`) and because internal g-trie nodes have multiple children, we are matching several different subgraphs at the same time and when we reach a leaf we already know the isomorphic class of the occurrence.

The algorithm starts by considering all vertices as possible candidates for the g-trie root node (lines 2-4). For each vertex it calls the recursive census function `COUNT` with a matched set containing only this vertex (line 4).

The `count()` function starts by computing a set of suitable candidate vertices that completely match the current g-trie node (line 6). This set is then traversed and recursively expanded through all possible tree paths (lines 7,11-12). If the node is a leaf this means it corresponds to a full subgraph and its frequency is incremented (lines 8-9).

The `matchingVertices()` function is pivotal in the efficiency of the search and fully utilizes the information stored in the g-trie to really limit and reduce the number of possible candidates. It starts by selecting the lowest degree vertex that must be a neighbor of the vertex being added (lines 14-15). From the neighbors of this vertex we take the ones that respect all connections to the ancestors and that do not break any symmetry breaking conditions (lines 16-18).

Given the space constraints we refer the reader to [21] for a more detailed view of this very efficient algorithm. For the purposes of our work, the main aspect to consider is that the bulk of the computation is spent inside the recursive `count()` function, which generates independent search tree branches that we will exploit resorting to parallelism.

**Algorithm 1** Computing the frequency of subgraphs of a g-trie $T$ in graph $G$. Adapted from [21].

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

## 4. PARALLEL G-TRIES ALGORITHM

Each call to the `count(`$T$`,` $V_{used}$`)` function is fully defined by its two arguments which indicate the position where we are at the g-trie ($T$) and the current set of network vertices that match to that position ($V_{used}$). With these two pieces of information, computation can be resumed without needing to know anything from the past. Furthermore, different calls are completely independent from each other. A pair ($T$, $V_{used}$) constitutes therefore our basic working unit.

At the upper level of the computation we have work units that correspond to all vertices being matched to the root node of the g-trie. We experimented with some static load balancing schemes that partition the work only at this level, but this is clearly not sufficient since the computation is highly unbalanced. Given the power law degree distributions of most real world complex networks, a few nodes may be responsible for a very significant percentage of subgraph occurrences, while others may have almost no occurrences at all. Experiments have shown cases where a single node, corresponding to a single work unit, would need more than 20% of the entire computation time, rendering any static division unfeasible for a scalable computation.

We needed a dynamic load balancing scheme, and our solution was to use an iterative MapReduce algorithm. During computation, a time threshold is used to force all the workers to stop and the remaining work is redistributed among all available computational resources for another iteration. For this methodology to work properly and in a scalable way we need several key ingredients: a good initial partition (that kickstarts the computation); a correct time granularity (stop too often and we will spend more time synchronizing and communicating than actually doing the subgraph counting; stop too late, and we will end up having idle workers that finished all their assigned work); a compact representation of unfinished computation (we need to actually communicate this to give computation to other workers); an efficient load redistribution mechanism (that allows work to be divided among workers for the next iteration). In the next sections we will detail how we approached these challenges.

## 4.1 Overview of our Parallel Algorithm

Initially, we divide the vertices in a round robin way, giving each worker an approximately equal number of nodes to process. After a fixed amount of time all the workers stop calculating, save and return their current state to the reducer, which collects these states and decides if the computation is finished. If not, the remaining work is divided between workers and a new MapReduce iteration is initiated.

As explained before, the value of the time threshold used for stopping highly affects the speedup. In order to maximize the parallel efficiency, we decided to use an adaptive threshold that dynamically changes during the execution according to what happened in the previous iteration.

We should also mention that we define two keywords regarding the work assigned to each worker. A *work unit* is a single state in the search tree from where the worker could start computing (as explained before), while a *work set* is a set of work units that is assigned to a worker.

Algorithm 2 details how we apply our strategy with g-tries. The controller (master/driver worker) starts by setting the initial time limit (line 5) and initial work sets (lines 6-7). Then, it launches the workers (lines 8-11). If the time limit is exceeded before the worker finishes its work, it saves its location and returns its current state which is a partial result (lines 26-29). Each state has two components, the found occurrences ($resArr$) and the remaining work in the corresponding work set. The Reducer collects those partial results, separates the two components, adds the occurrences to the final result and constructs the total remaining work. If we still have work left to do, the driver distributes this work (line 12), adjusts the threshold (line 13) and launches the workers with the new work sets (lines 8-11).

**Algorithm 2** Computing the frequency of $k$-subgraphs of a g-trie $T$ in graph $G$ using $w$ workers using MapReduce

```
 1: procedure MAIN($T, G, w$)
 2:     $result \longleftarrow$ A g-trie that will contain the total result.
 3:     $verticesSets \longleftarrow$ The list of work-sets for workers, $[w]$.
 4:     $partialResult \longleftarrow$ The result of each worker.
 5:     $timeLimit = k^3 * (G.numVertices)^2$;  $ctr = 0$
 6:     while $ctr < G.numVertices$ do
 7:         $verticesSets[ctr \mod w].add(ctr)$;  $ctr++$
 8:     while $verticesSets \neq \emptyset$ do
 9:         $partialResult$ = VERTICESSETS.MAP(vSet) {
10:           COUNTALLMAPPER($T, vSet, timeLimit$)}
11:         $result = partialResult$.REDUCE($partRes1, partRes2$)
12:         $verticesSets = redistribute(remainingWork)$
13:         $timeLimit = adjustThreshold(nIMappers, avgIdleTime)$
14:     Print $result$
15: procedure COUNTALLMAPPER($T, G$)
16:     $currentState = null$
17:     for all vertex $v$ of $G$ do
18:         for all child $c$ of $T.root$ do
19:             $currentState$ = COUNT($c$, $\{v\}$, $currentState$)
20:             if Time Limit exceeded then
21:                 return $currentState$
22:     return $currentState$
23: procedure COUNT($gN, V_{used}, currentState, resArr$)
24:     $V \leftarrow$ MATCHINGVERTICES($gN, V_{used}$)
25:     for all vertex $v$ of $V$ do
26:         if Time Limit exceeded then
27:             $remainingV$ = the unexplored vertices in $V$
28:             $currentState$ = SAVESTATE($remainingV, hLable,$
      $currentMatch, trieID, currentState$)
29:             return $currentState$
30:         if $gN.isLeaf$ then
31:             $resArr[gN.id]++$
32:         else
33:             for all child $c$ of $gN$ do
34:                 COUNT($c, V_{used} \cup \{v\}, currentState, resArr$)
35: procedure SAVESTATE($remainingV, hLable, currentMatch,$
      $trieID, currentState$)
36:     $state = trieID + hLable + currentMatch + remainingV$
37:     $currentState$ += $state$
38:     return $currentState$
```

## 4.2 Saving the State

When the time threshold is reached, every mapper that has not finished processing stops computing and saves its search state. The goal is to save the state of the recursive work by capturing the stack contents in an efficient way. Furthermore, before returning its state, each worker divides its own remaining work in a number of work sets equal to the amount of mappers. By doing this, we make sure that even if all workers have finished their work except one, all of them will still have work to do in the next iteration.

The function `count()` of Algorithm 2 contains two cycles: one loops over all possible vertex candidates and the other loops over all children of the corresponding g-trie node. In order to save the state in each depth we need to store the position in both cycles, and we do so by using simple integer labels. Each vertex in the graph already has a label to define it. We also assign an id to each g-trie node before we start the computation, and since the g-trie does not change during computation, we can use it as a label. Inside each recursive level, we attribute unexplored vertices to the several work sets in a round-robin fashion.

## 4.3 Resuming the Work

As we said before, each saved working unit has enough information to continue its work independently. However, we need a procedure to do the bridge between the saved working unit and the recursive counting function (**count**). This procedure is shown in Algorithm 3. It iterates through the remaining vertices stored in the saved work unit and continues counting the occurrences by calling the original census function (lines 10-11).

## 4.4 Adaptive Threshold Mechanism

Initially, we set the threshold value to $G.numVertices^2 \times motifSize^3$ nanoseconds (line 5). The intuition is that larger network and motif sizes induce more subgraph occurrences. Hence, this will result in larger computation times and therefore the granularity should be bigger.

After every iteration the threshold is re-adjusted according to the number of workers who waited without work in the previous iteration and how much time they were idle.

In the case where no workers went idle in the previous iteration we **increase** the threshold by 20%.

In the other case, when some worker were idle during the previous iteration, we **decrease** the threshold:

**Algorithm 3** `G-Tries`: resuming the work from a saved state.

```
1: procedure RESUMEWORK($T, V_{used}, remainingVertices, currentState$)
2:     for all vertex $v$ of $remainingVertices$ do
3:         if Time Limit exceeded then
4:             $remainingV$ = the unexplored vertices.
5:             $currentState$ = SAVESTATE($remainingV, hLable, currentMatch, trieID, currentState$)
6:             return $currentState$
7:         if $T.isLeaf$ then
8:             $T.frequency{+}{+}$
9:         else
10:            for all child $c$ of $T$ do
11:                COUNT($c, V_{used} \cup \{v\}, currentState$)
```

$$newT = oldT - \frac{oldT * numIdleWorkers}{totNumWorkers} - (20\% * avgWaittedTime)$$

Again, the intuition is that when all workers still have work units to process, we should increase the granularity, so as to reduce the amount of time spent unnecessarily synchronizing from one iteration to the other. Similarly, if there are idle workers, we should decrease the granularity so that we can redistribute work earlier and keep all workers busy. We did systematic empirical tests to find the best values for the parameters that affected this mechanism and 20% was chosen after being **the value that maximized average efficiency over most of tests**. We omit the details of this experiment due to the space constraints of this paper.

## 5. EXPERIMENTAL EVALUATION

In this section we present empirical data obtained by running our parallel methodology on a diverse set of representative complex networks. First, we describe the computational environment and the networks used. Next, we provide a parallel performance evaluation, showing the speedups we obtained.

### 5.1 Computational Environment

We ran all of our tests on a 64-core machine, consisting of four 16-core AMD Opteron 6376 processors at 2.3GHz with a total of 252GB of memory installed. Each 16-core processor is split in two banks of eight cores, each with its own 6MB L3 cache. Each bank is then split into sets of two cores sharing a 2MB L2 and a 64KB L1 instruction cache. A 16KB L1 data cache is dedicated to each core. Because L2 cache is shared between pairs of cores, **we only had access to 32 truly independent cores**, and this is reflected in the results we obtained.

All code was developed in Java and compiled using Maven 3.3.9, inside the Spark framework. Moreover, the used time unit is the second.

### 5.2 Complex Networks

For the purpose of testing our work and showing its general applicability, we chose a diverse set of networks from various fields of application. Table 1 gives an overview of their topological characteristics and indicates its source.

### 5.3 Baseline Results

| **Network** | **Src** | **Type** | $\lvert V(G)\rvert$ | $\lvert E(G)\rvert$ | **Avg. Degree** |
|---|---|---|---|---|---|
| `polblogs` | [16] | communic. | 1,491 | 19,022 | 12.8 |
| `foldoc` | [5] | semantic | 13,356 | 120,238 | 9.0 |
| `september11` | [5] | semantic | 13,314 | 243,447 | 18.3 |
| `gnutella` | [13] | internet | 8,717 | 31,525 | 3.6 |
| `company` | [5] | communic. | 8,497 | 6,724 | 0.8 |
| `facebook` | [13] | social | 4,039 | 88,234 | 21.9 |
| `wikivote` | [13] | wikipedia | 7,115 | 103,689 | 14.6 |
| `neural` | [16] | biological | 297 | 2,345 | 7.9 |
| `metabolic` | [3] | biological | 453 | 2,025 | 4.7 |
| `netscience` | [16] | collaboration | 1,589 | 2,742 | 1.7 |

**Table 1:** The set of representative real networks used for experimental evaluation.

Since the original implementation of g-tries is written in C++ and we rewrote the algorithms in Java (so as to use them within Spark) our first task was to compare our sequential port with the original algorithm and we found out that our implementation was on average twice as slow. Furthermore, we also compared the time of our parallel approach using only one core when compared to the sequential version, in order to measure the overhead introduced by using MapReduce and Spark. Results have shown that the overhead is on average around 100%.

Given that we wanted to study the parallel efficiency of our algorithm, we needed to have computation times that were large enough to justify parallelization, but at the same time were small enough to be feasible to compute in due time. We computed full size $k$ undirected subgraph census for all the networks and chose a suitable size $k$. Table 2 summarizes the used sizes for each network. The growth rate gives an indication on how much more time it takes to compute a subgraph of size $k+1$ when compared to $k$, providing an estimation of the required computation time for larger subgraph sizes and at the same time showcasing the heterogeneity of the topologies of the networks we used. We should also mention that the actual number of subgraph occurrences of the census referred to in the table are in the order of $10^9$ to $10^{10}$.

| **Network** | **Sub. Size** | **Total Time (s)** | **Growth Rate** | |
|---|---|---|---|---|
| | | | AVG | STD |
| `polblogs` | 5 | 2,662 | 57.5 | 36.1 |
| `foldoc` | 5 | 6,346 | 83.0 | 40.3 |
| `septemper11` | 4 | 2,725 | 305.6 | 3.9 |
| `gnutella` | 6 | 1,964 | 16.5 | 12.1 |
| `company` | 5 | 782 | 59.4 | 74.0 |
| `facebook` | 5 | 4,899 | 88.8 | 74.7 |
| `wikivote` | 4 | 737 | 56,055.7 | 96,449.4 |
| `neural` | 7 | 13,215 | 2,690.7 | 5884.1 |
| `metabolic` | 6 | 2,183 | 552.2 | 1087.3 |
| `netscience` | 9 | 2,851 | 452.3 | 1062.5 |

**Table 2:** Baseline execution times with one core for undirected subgraph census.

### 5.4 Parallel Results

The results we obtained are very promising for a MapReduce approach, with a consistent and almost linear speedup up to 32 cores, with an average speedup close to 25 and an average efficiency close to 80%. We can see a small degradation when we move to 64 cores, which can be explained by the limitation imposed by the hardware we used, which only provides 32 independent cores (with each pair of cores sharing part of their caches). We believe that when using truly independent cores the trend from 32 cores would continue to scale up consistently. A more detailed view of the obtained speedups can be seen in Table 3.

The differences between different networks are mainly due to very different topologies and execution times. A detailed analysis of the results shows that in the general case the better speedups are obtained for the cases in which the computation time is higher. This means that we are cutting the needed time in the use cases that most need it and means that we could potentially scale well for even bigger cases. For instance, in the *company* network, the computation time for 32 cores is already only 39 seconds and it is hard to improve upon this. By contrast, in the *neural* network, 32 cores take 506 seconds and so there is still room for parallel improvement when doubling the number of processors.

| **Network** | **Sub. size** | **#workers: speedup** | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | 4 | 8 | 16 | 32 | 64 |
| `polblog` | 5 | **1.9** | **3.6** | **7.2** | **13.6** | **26.0** | **41.8** |
| `foldoc` | 5 | **1.9** | **3.6** | **7.1** | **14.2** | **26.4** | **42.1** |
| `septemper11` | 4 | **1.9** | **3.7** | **6.9** | **12.7** | **24.1** | **36.0** |
| `gnutella` | 6 | **1.9** | **3.8** | **6.7** | **12.5** | **24.3** | **40.1** |
| `company` | 5 | **1.9** | **3.8** | **7.1** | **12.2** | **20.1** | **25.2** |
| `facebook` | 5 | **1.8** | **3.4** | **7.1** | **13.9** | **25.4** | **36.6** |
| `wikivote` | 4 | **1.9** | **3.8** | **7.1** | **12.5** | **20.6** | **25.1** |
| `neural` | 7 | **1.9** | **3.7** | **7.3** | **13.9** | **26.1** | **43.2** |
| `metabolic` | 6 | **1.9** | **3.6** | **7.1** | **13.9** | **26.6** | **40.0** |
| `netscience` | 9 | **1.9** | **3.9** | **6.8** | **11.3** | **19.0** | **25.9** |

**Table 3:** Speedups obtained with our parallel approach when computing undirected subgraph census.

For the purpose of showing that our strategy is general and could be applied when searching directed subgraphs, we made some additional tests in which the direction of edges was kept if the original dataset had it. Table 4 summarizes the obtained results, which were very similar to the undirected case.

| **Network** | **Sub. size** | **#workers: speedup** | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | 4 | 8 | 16 | 32 | 64 |
| `foldoc` | 5 | **1.9** | **3.4** | **5.4** | **13.0** | **26.1** | **43.6** |
| `company` | 5 | **1.8** | **3.8** | **6.6** | **13.5** | **23.3** | **31.4** |
| `wikivote` | 4 | **1.9** | **3.4** | **5.4** | **13.0** | **29.6** | **45.5** |

**Table 4:** Speedups obtained with our parallel approach when computing directed subgraph census.

Figure 4 summarizes our results, by plotting the average speedup of all tested networks in both the directed and undirected cases. We can clearly see that we achieve close to linear speedups up to 32 cores and using 64 cores case we still obtain considerable speedup only limited by the used hardware.

As a result, with such speedups, some computations which were very hard to run will be possible and that gives the user the ability to search for larger subgraphs, which is very important toward extracting new information. For instance, we computed a 5-census on the **`wikivote`** network using the sequential version and it took around two days. Using our parallel algorithm and 64 workers, it only took 1.2 hours.

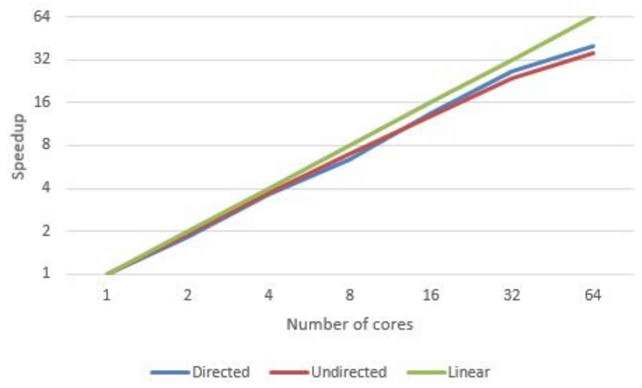

**Figure 4:** Average speedup for all tested networks.

## 5.5 Comparison with Competing Algorithms

There are not too many algorithms that can be directly compared to our methodology, given the fundamental differences pointed out in section 2.3. Perhaps the most comparable work is the one from Verma et al. [24], which also employs iterative MapReduce, although it uses the much slower ESU as the base sequential algorithm. Since we could not find any implementation online, we tried to run our algorithm on a network that was referenced in their work and we compared the obtained results.

Their second best speedup was using a protein protein interaction (PPI) network which has **2,365** nodes, and searching for subgraphs of size **7** in this network. The processors in our machine are slower (*2.3 GHz* versus *3.4 GHz*). However, we used the same number of cores towards making the comparison as fair as possible. Table 5 shows the obtained results.

| **Method** | **Sequential** | **Parallel** | **Speedups** | **Efficiency** |
|---|---|---|---|---|
| `ESU` | 172,800 | 19,686 | 8.78 | ≈ 15.7% |
| `G-Tries` | 9,539 | 266 | 35.86 | ≈ 64% |

**Table 5:** Comparison between our approach and Verma et al. [24] on a PPI network.

As shown in the table, our speedups and efficiency are higher. Our g-tries Java implementation is already 18 times faster using one processor (due to the underlining base sequential algorithm). Moreover, our speedup and efficiency are much higher. In conclusion, to do this test using approximately the same machine, their parallel algorithm took 5.5 hours and our parallel approach took 4.4 minutes. We are aware that this is a very superficial and incomplete comparison, but we wanted to showcase the potential of our approach when compared to a very recent published work geared towards the same computational task and using the same parallel programming model.

## 6. CONCLUSION

Complex networks are used in a wide range of artificial and natural systems. The detection of small patterns in these networks leads to a better understanding of their structure and functionality. This operation is called subgraph search and has been applied to networks in many fields. However, it is a computationally hard problem and because of that its application is limited by the size of the pattern being searched and the size of the network.

For the purpose of decreasing those limitations, we presented a parallel MapReduce strategy that speeds up subgraph census, using the state-of-the-art g-tries data structure as a sequential basis. We ensure load balancing between the workers by using an adaptive time threshold and an efficient work-sharing mechanism where workers can stop, save and resume their computations from anywhere in the search tree.

Our algorithm was tested in several representative networks from various fields and presented near-linear speedup up to 32 cores, providing what we believe is the fastest and most scalable method for subgraph counting within the MapReduce programming model. The promising speedups and the availability of MapReduce on cloud providers allow the exploration of larger subgraphs in bigger networks.

**Acknowledgments** This work is supported by ERDF through the COMPETE Programme within project POCI-01-0145-FEDER-006961, by national Funds through FCT within projects Reminds/UTAP-ICDT/EEI-CTP/0022/2014 and UID/EEA/50014/2013. The work is also within FourEyes, a research line within project "TEC4Growth/NORTE-01-0145-FEDER-000020".

## 7. REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 62–73. IEEE, 2013.

[2] D. Aparício, P. Ribeiro, and F. Silva. Parallel subgraph counting for multicore architectures. In *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 34–41. IEEE, 2014.

[3] A. Arenas. Alex arenas - network datasets, 2014. http://deim.urv.cat/~alexandre.arenas/data/welcome.htm .

[4] A.-L. Barabasi. *Network science*. Cambridge University Press, 2016.

[5] V. Batagelj and A. Mrvar. Pajek datasets, 2006. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[6] C. Boyd. Data-parallel computing. *Queue*, 6(2):30–39, 2008.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[8] L. d. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242, 2007.

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

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

[11] S. Khakabimamaghani, I. Sharafuddin, N. Dichter, I. Koch, and A. Masoudi-Nejad. Quatexelero: an accelerated exact network motif detection algorithm. *PloS one*, 8(7):e68073, 2013.

[12] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974–985, June 2015.

[13] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[14] B. D. McKay and A. Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

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

[16] M. Newman. Network datasets, 2010. http://www-personal.umich.edu/~mejn/netdata/.

[17] P. Paredes and P. Ribeiro. Towards a faster network-centric subgraph census. In *Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on*, pages 264–271. IEEE, 2013.

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

[19] P. Ribeiro and F. Silva. Efficient subgraph frequency estimation with g-tries. In *International Workshop on Algorithms in Bioinformatics*, pages 238–249. Springer, 2010.

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

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

[22] P. Ribeiro, F. Silva, and L. Lopes. Efficient parallel subgraph counting using g-tries. In *2010 IEEE International Conference on Cluster Computing*, pages 217–226. IEEE, 2010.

[23] S. Shahrivari and S. Jalili. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing*, 97(11):1101–1120, 2015.

[24] V. Verma, P. P. Kwon, and W. Kim. Iterative hadoop mapreduce-based subgraph enumeration in network motif analysis. In *Cloud Computing, IEEE 8th International Conf. on*, pages 893–900. IEEE, 2015.

[25] S. Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions oon Computational Biology and Bioinformatics (TCBB)*, 3(4):347–359, 2006.

[26] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. A. Kumar, and M. V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 390–401. IEEE, 2012.