2010 IEEE International Conference on Cluster Computing

Efficient Parallel Subgraph Counting using G-Tries

Pedro Ribeiro, Fernando Silva, Luís Lopes
CRACS & INESC-Porto LA,
Faculdade de Ciências, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
Email: {pribeiro, fds, lblopes}@dcc.fc.up.pt

***Abstract*—Finding and counting the occurrences of a collection of subgraphs within another larger network is a computationally hard problem, closely related to graph isomorphism. The subgraph count is by itself a very powerful characterization of a network and it is crucial for other important network measurements. G-tries are a specialized data-structure designed to store and search for subgraphs. By taking advantage of subgraph common substructure, g-tries can provide considerable speedups over previously used methods. In this paper we present a parallel algorithm based precisely on g-tries that is able to efficiently find and count subgraphs. The algorithm relies on randomized receiver-initiated dynamic load balancing and is able to stop its computation at any given time, efficiently store its search position, divide what is left to compute in two halfs, and resume from where it left. We apply our algorithm to several representative real complex networks from various domains and examine its scalability. We obtain an almost linear speedup up to 128 processors, thus allowing us to reach previously unfeasible limits. We showcase the multidisciplinary potential of the algorithm by also applying it to network motif discovery.**

***Keywords*-Parallel Algorithms, Adaptive Load Balancing, Complex Networks, Graph Mining, G-Tries**

I. Introduction

Complex networks are everywhere [1]. They can describe a very wide range of natural and artificial systems. Understanding their underlying topology, and how they can be modeled and characterized has a broad multidisciplinary applicability [2]. In order to extract useful features from these networks, a large number of metrics were developed [3].

Finding and counting subnetworks within the whole global network is one very important associated task. We could be interested in discovering the most frequent patterns, which can for example be used for network indexing [4]. We might find useful the frequency of all subnetworks, in order to characterize the network [5]. Or we could want to discover subnetworks that occur more often than in similar randomized networks [6].

Computing the frequency of one or more subnetworks is a computationally *hard* problem since we are basically dealing with graph isomorphism [7]. For special cases of networks and subnetworks polynomial solutions exist [8]. When searching only for the most frequent subgraphs, we may me able to narrow the search space [9]. But for the general case, possibly with infrequent subgraphs, the problem remains completely exponential in nature.

The methods currently used for the general subgraph counting problem typically opt for one of two opposite approaches: either we enumerate all subnetworks of a desired size and then compute which ones are isomorphic [6], [10] (*network-centric*); or by knowing the subnetworks to search, we compute the individual frequency of each one by running an efficient subgraph matching procedure for each subnetwork [11] (*subgraph-centric*).

By knowing the subnetworks that we are looking for, we could try to solve the problem by using an intermediate approach. Instead of having to enumerate all subnetworks (and only after do the isomorphism tests) or matching one at a time, we could at the same time try to match all the desired subnetworks. G-tries are a very recent data structure that can be used precisely to this end [12]. They take advantage of common topologies to create a tree that represents a set of subgraphs, with which we simultaneously search multiple subnetworks. This leads to considerable speedups when compared to the previous methods, up to one hundred times faster for some networks.

The current methods are also almost all sequential in nature. Considering the computational tractability, we feel that resorting to parallel algorithms to speedup the computation can have a strong multidisciplinary impact. More than just computing faster, we can reach subgraph and graph sizes that were previously not reachable for efficiency reasons.

Our main contribution in this paper is an efficient parallel algorithm for the general subgraph counting problem, that uses g-tries as the underlying data structure. We extend and parallelize the counting procedure with receiver-initiated dynamic load balancing [13], paving the way for efficient and scalable counting for any types of subgraphs and graphs.

The remainder of this paper is organized as follows. Section II establishes a network terminology, formally defines the problem we are tackling and overviews the related work. Section III describes the g-trie data structure and details the parallel algorithm we developed. Section IV discusses the results obtained for a set of representative networks and gives an example of a practical application in the discovery of network motifs. Section V concludes the paper, with comments on the results and possible future work.

II. Preliminaries

A. Graph terminology

In order to have a well defined and coherent network terminology throughout the paper, we first review the main concepts and introduce some notation that will be used on the following sections.

A network can be modeled as a *graph* G composed of the set $V(G)$ of *vertices* or *nodes* and the set $E(G)$ of *edges* or *connections*. The *size* of a graph is the number of vertices and is indicated as $|V(G)|$. A k-graph is a graph of size k. Every edge is composed by a pair of two *endpoints* in the set of vertices. This pair is ordered in the case of a *directed* graph, in opposition to *undirected* graphs where edges do not express direction. The *neighborhood* of a vertex u in a graph G, is a subgraph, denoted as $N(u)$, composed by the set of all other nodes v of G such that (u, v) or (v, u) belong to $E(G)$. All vertices are assigned consecutive integer numbers starting from 0, and the comparison $v < u$ means that the index of v is lower than that of u. The adjacency matrix of a graph G is denoted as G_{Adj}, and $G_{Adj}[a][b]$ represents a possible edge between vertices with index a and b.

A *subgraph* G_k of a graph G is a graph of size k in which $V(G_k) \subseteq V(G)$ and $E(G_k) \subseteq E(G)$. This subgraph is said to be *induced* if for any pair of vertices u and v of $V(G_k)$, (u, v) is an edge of G_k if and only if (u, v) is an edge of G, that is, a vertex set of the subgraph has all the edges that the same vertex set has in the complete graph G.

A *mapping* of a graph is a bijection where each node is assigned a value. Two graphs G and H are said to be *isomorphic*, denoted as $G \sim H$, if there is a one-to-one mapping between the vertices of both graphs where two vertices of G share an edge if and only if their corresponding vertices in H also share an edge.

B. The General Subgraph Counting Problem

Here we formalize the problem we are tackling, which is computing the frequency of a determined set of subgraphs within a larger graph.

Definition 1 (General Subgraph Counting Problem): *Given a set of subgraphs S_G and a graph G, determine the exact count of all induced occurrences of subgraphs of S_G in G. Two occurrences are considered different if they have at least one node or edge that they do not share. Other nodes and edges can overlap.*

This definition is very flexible and can be applied to any type of graphs, whether they are directed or undirected, colored or not and weighted or not. The second part of the definition contains the most widely used concept for distinguishing two occurrences of a subgraph. Note that this has a direct impact on the number of the subgraphs and on the problem's tractability, since there is no downward closure property, i.e., a subgraph may appear more times than a subgraph contained in it [14].

Note also that this problem is conceptually very different from *frequent subgraph discovery* [15] (FSD), which derives from the *frequent itemset problem* [16]. In FSD we look for subgraphs that occur at least a pre-determined number of times (in opposition to really computing the frequency) in a minimum number of elements of a set of graphs (in opposition to only one single original graph). Therefore, the algorithmic techniques used are different, with more focus on creating increasingly larger candidate graphs using the *a*-priori principle [17], and are not applicable to our problem.

C. Related Work

To our best knowledge, the ESU algorithm [10] presents the most efficient full enumeration of k-subgraphs (*network-centric* approach), i.e., computing a subgraph census. Grochow and Kellis [11] show how to efficiently count the occurrences of an individual subgraph (*subgraph-centric* approach).

All the aforementioned methods are sequential in nature and parallel work on general subgraph counting is still very scarce. Wang et al. [18] provide a complete census with static load balancing and limited scalability study up to 32 processors. Ribeiro et al. [19] also parallelize a complete census based on the ESU algorithm with dynamic load balancing and almost linear speedup up to 128 processors. Schatz et al. [20] parallelize the Grochow and Kellis approach, using a master-worker load balancing scheme of several individual queries, and present an almost linear speedup up to 64 processors on the single network studied. They also try to parallelize an individual subgraph query but show very limited scalability tests.

The main difference in our work is that we use g-tries as the underlying data-structure and therefore we have a new, potentially more efficient, start point. G-tries associated sequential methods and performance comparison to other approaches can be seen in [12].

III. Parallel Algorithm

A. The G-Trie Data Structure

A g-trie is a multiway tree able to store a collection of subgraphs, that is, a set of abstract graph "patterns". Each tree node contains information about a single subgraph vertex and its connections to the ancestor nodes. Descendants of a tree node share a common topology and a path from the root to a node defines one single subgraph. Figure 1 gives an example of a g-trie with 6 undirected subgraphs.

Each g-trie node must store information about the connections to ancestor nodes and to itself. If the subgraphs are undirected, we can do that by storing the respective adjacency matrix row (up to that node position), which will effectively result in only half of the full symmetrical adjacency matrix being stored in the g-trie. If the subgraphs

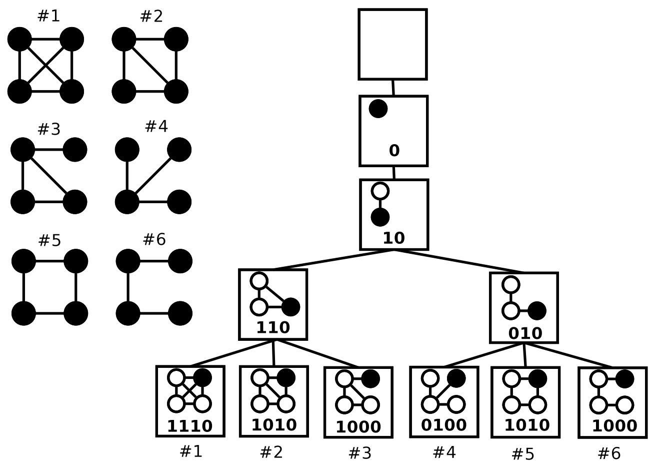

Figure 1. A g-trie representing a set of 6 undirected subgraphs. Each g-trie node adds a new vertex (in black) to the already existing ones in the ancestor nodes (white vertices). The connections to these nodes are represented by a sequence of numbers indicating the corresponding adjacency matrix row.

are directed, then we must also store the respective matrix column, as well as the row, in order to specify in and out edges.

Note that the g-trie root node must be empty, because there are two possible direct descendant nodes: a vertex with or without a connection to itself.

In order to obtain an unique g-trie representation for a set of subgraphs and guarantee that a specific subgraph always leads to the same tree path, we use a canonical adjacency matrix. Among the many possibilities, one can for example use the lexicographically larger, favoring the occurrence of more common substructures with higher degree nodes appearing in lower depth levels.

This capability of identifying common sub-topologies is the strength of a g-trie. Not only can we compress the information by avoiding redundant storage, but when we are matching a specific node in the g-trie we are matching at the same time all possible descendant subgraphs stored in the g-trie.

In order to avoid subgraph symmetries, g-tries also store symmetry breaking conditions of the form $a < b$ indicating that the vertex in position a should have a graph index smaller than vertex in position b (refer to [12] for more details). Using these conditions, we can find each subgraph only once.

In order to avoid ambiguities in the description, from now on we will use the term node to refer to the g-trie tree nodes, and vertex to refer to a node in the stored subgraphs. Given a g-trie node T, we will use $T.vertex$ to refer the new vertex of that node (represented in black in Figure 1, and $T.in[i]$ and $T.out[i]$ to refer to the boolean value of the new vertex having respectively an ingoing or outgoing connection to the vertex with index i, i.e., the new node represented in the ancestor of depth i. Note that if the g-trie stores undirected subgraphs, then $T.in[i] = T.out[i]$. We will also use $T.cond$ to denote the set of conditions that break symmetries for the descendant nodes that correspond to a full subgraph. $T.root$ denotes the g-trie root node and $T.isGraph$ indicates if the node stores a full subgraph (in Figure 1 this corresponds to all leaf nodes).

B. Adapted Sequential Algorithm

In order to conceive a parallel approach we started by looking at our g-trie serial matching and modified it in a way suitable for parallelization. Our aim was to create a framework where given some part of the whole computation, we would either be able to compute it completely or divide it in similar independent smaller pieces of computation.

The algorithm in Figure 2 details the initial adaptation of our sequential algorithm, now tailored to be later adapted for parallel execution.

```
1:  **procedure** MATCHALL(T, G)
2:      **for all** children $c$ of $T.root$ **do**
3:          MATCH($c$, $\emptyset$)
4:  **procedure** MATCH($T, V_{used}$)
5:      $V$ = MATCHINGVERTICES($T, V_{used}$)
6:      **for all** node $v$ of $V$ **do**
7:          **if** $T.isGraph$ **then** FOUNDMATCH()
8:          **for all** children $c$ of $T$ **do**
9:              MATCH($c, V_{used} \cup \{v\}$)
10: **function** MATCHINGVERTICES($T, V_{used}$)
11:     **if** $V_{used} = \emptyset$ **then**
12:         $V_{cand} := V(G)$
13:     **else**
14:         $V_{conn} = \{v : v = V_{used}[i], T.in[i] \vee T.out[i]\}$
15:         $m := m \in V_{conn} : \forall v \in V_{conn}, |N(m)| \leq |N(v)|$
16:         $V_{cand} := \{v \in N(m) : v \notin V_{used}\}$
17:     $Vertices = \emptyset$
18:     **for all** $v \in V_{cand}$ **do**
19:         **if** $\forall i \in [1..|V_{used}|]$:
20:             $T.in[i] = G_{Adj}[V_{used}[i]][v] \wedge$
21:             $T.out[i] = G_{Adj}[v][V_{used}[i]]$ **then**
22:             **if** $\exists C \in T.cond : V_{used} + v$ respects $C$ **then**
23:                 $Vertices = Vertices \cup \{v\}$
24:     **return** $Vertices$
```

Figure 2. Algorithm for matching subgraphs of g-trie T in graph G

The basic idea of the algorithm is to try to find a set of vertices (V_{used}) that match to a path in the g-trie and therefore correspond to an occurrence of the subgraph represented by that path. We use the information stored in the g-trie to heavily constrain and limit the search.

We start with the g-trie root children nodes and call the recursive procedure `match()` with an initial empty matched set (lines 2 and 3). The later procedure starts by creating a set of vertices that fully match the current g-trie node (line 5). We traverse that set (line 6) and recursively try to expand it through all possible paths (lines 8 and 9). If the node corresponds to a full subgraph, then we found an occurrence of that subgraph (line 7).

Generating the set of matching vertices is done in the `matchingVertices()` procedure. The efficiency of the algorithm depends heavily on the above mentioned constraints, since they help reduce the search-space. To generate the matching set, we start by creating a set of candidates (V_{cand}). If we are at a root children, then all graph vertices are viable candidates (lines 11 and 12). If not, we select from the already matched vertices that are connected to the new vertex (line 14) the one with the smallest neighborhood (line 15), reducing the possible candidates (line 16). Then, we traverse the set of candidates (line 18) and if one respects all connections to ancestors (lines 19 to 21) and respects at least one set of symmetry breaking conditions for a possible descendant subgraph (line 22), we add it to the set of matching vertices (line 23).

C. Parallel Subgraph Counting

This sequential algorithm produces a tree shaped search space with each node being a call to `match`(T, V_{used}). A crucial aspect is that all these calls are independent from each other. A pair (T, V_{used}) uniquely identifies where we are in the search and we can continue from that point without knowledge of what may have been computed. From now on we call such a pair a *work-unit*. The processing of one work-unit may originate other work-units (if we are able to progress to a higher depth on the g-trie and add other vertex to V_{used}) or may not result in new work (if we are in a g-trie leaf or if no suitable graph vertex candidate is found).

In order to parallelize our search we need to distribute the work-units among all processors. The problem is that the search tree is highly unbalanced and the execution time of each work-unit varies significantly. This makes it very hard to use a pre-determined static allocation scheme, because approximating the execution time cost of a work-unit can be as hard as computing the work-unit itself and, therefore, we opted for a dynamic load balancing strategy.

Our target parallel programming model is distributed memory with message passing. We opted for a receiver-initiated scheme [13] in order to cope with a heavy system load. The algorithm Figure 3 gives a global simplified view of our approach that we detail later.

Basically, each processor starts by choosing its initial share of work (line 2) and then keeps processing it (line 4) until the whole global computation is completed (line 3). While doing this, after a determined threshold is reached (line 12), it checks for incoming messages from other processors (13). If a work request message was received (line 14), the recursive computation is stopped and the search state stored (line 15). Then it uses this state to divide the current search into two different sets of work units (line 6), sending one to the requesting processor (line 7) and keeping one to itself, in order to continue the computation. If there were no work request messages, the recursive search is completed, ending in a situation with no more work-units to compute,

```
1:  procedure PARALLELCOUNT
2:      $W$ = GETINITIALWORK()
3:      while NOTFINISHED() do
4:          RECURSIVEPROCESS($W$)
5:          if RECEIVEDWORKREQUEST() then
6:              $(W, W_2)$ = DIVIDEWORK()
7:              SENDWORK($W_2$)
8:          if $W = \emptyset$ then
9:              $W$ = ASKFORMOREWORK()
10:     AGGREGATERESULTS()
11: procedure RECURSIVEPROCESS($W$)
12:     if CHECKMESSAGESTHRESHOLD() then
13:         CHECKMESSAGES()
14:     if RECEIVEDWORKREQUEST() then
15:         stop and store recursive computation
16:     else
17:         keep doing recursive search
```

Figure 3. Simplified view of parallel algorithm for subgraph counting

and therefore the processor starts looking for unprocessed work-units from another processor (line 9). Finally, after all work is done, we aggregate the distributed results in order to produce the desired output (line 10).

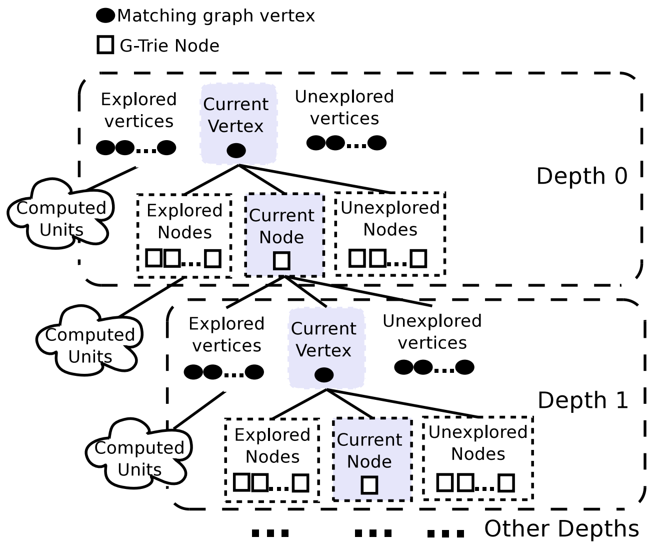

Figure 4. The state of the `match()` procedure frozen at a given time. In gray we illustrate the exact subgraph being matched, corresponding to the current position in the two main cycles of the recursive procedure

The main core of the search is the recursive procedure, which will correspond to the `match()` procedure of the sequential algorithm. A crucial extension for our parallel approach is that we must be able to stop and store the state of that recursive computation. For this we must capture the stack contents and we want to do it in an efficient way. Figure 4 depicts the recursive state of the computation at any given time. Note that `match()` (fig. 2) revolves around two cycles: one enumerates all possible matching vertices (line 6) and the other around all possible children

D	-	Larger recursive depth
V_i	-	currently explored graph vertex in depth i
N_i	-	currently explored g-trie node in depth i
U_i	-	number of unexplored vertices in depth i
UV_i^j	-	j-th unexplored vertices in depth i

D	V_0	V_1	$\dots$	V_{D-1}	N_0	N_1	$\dots$	N_{D-1}	U_1	UV_1^1	UV_1^2	$\dots$	$UV_1^{U^1}$	U_D	UV_D^1	UV_D^2	$\dots$	$UV_D^{U^D}$			

Figure 5. The structure of a work array representing completely the recursive search state.

of the corresponding matching g-trie node (line 8). If we freeze time, the search position will therefore be defined by knowing the position where we are at each of these two cycles in all depths.

In order to store this state, we need to save the cycle position (i.e., the current node and vertex) for all depths of the recursive procedure. In the case of a g-trie node, by knowing the current node one can instantly know the remaining nodes to explore, since the g-trie is fixed. In the case of the vertices, we must explicitly store the unexplored vertices, because they are dynamic and correspond to real computation work done in `matchingVertices()` (lines 10 to 24 of fig. 2). We can univocally identify each g-trie node by a single integer number, and the same can be done with each graph vertex. With this in mind, we create a compact array structure that encapsulates everything that we need in order to later resume the search. This array is depicted in Figure 5 and we will call it a *work array*.

The algorithm in Figure 6 details our adaptation of the `match()` procedure so that it is able to stop and store the recursion whenever a request for work arrives. In this case, it builds the correspondent work array and stops the search.

```
1:  procedure PARALLELMATCH($T, V_{used}$)
2:      if CHECKMESSAGESTHRESHOLD() then
3:          CHECKMESSAGES()
4:      $V$ = MATCHINGVERTICES($T, V_{used}$)
5:      for all node $v$ of $V$ do
6:          if RECEIVEDWORKREQUEST() then
7:              $state$.ADD(remaining nodes of $V$); break
8:          if $T.isGraph$ then FOUNDMATCH()
9:          for all children $c$ of $T$ do
10:             if RECEIVEDWORKREQUEST() then
11:                 $state$.ADD(current children $c$); break
12:             PARALLELMATCH($c, V_{used} \cup \{v\}$)
```

Figure 6. Parallel version of `match()` procedure

Lines 4, 5, 8, 9 and 12 are the same as in the sequential algorithm. What changes is that now we keep checking for messages as explained before (lines 2 and 3). When a request for work arrives, we stop making recursive calls, break all cycles, and build the work array (lines 6, 7, 10 and 11).

The algorithm of Figure 7 shows how to resume a given work array W, assuming the notation given in Figure 5. We also assume that the child nodes of a g-trie node are ordered and that by *bigger siblings* we mean the g-tries nodes that share the same direct ancestor node, i.e., the same "father", and that are bigger in that order context.

```
1:  procedure RESUMEWORK($W$)
2:      $V_{used} = \{V_0, \dots, V_{D-1}\}$
3:      for $i$: $D-1$ down to 0 do
4:          $N_{remaining} = N_i \cup$ BIGGERSIBBLINGS($N_i$)
5:          for all $C$ in $N_{remaining}$ do
6:              if RECEIVEDWORKREQUEST() then
7:                  $state$.ADD(remaining children); break;
8:              PARALLELMATCH($C, V_{used}$)
9:          for $j$: 1 to $U^i$ do
10:             if RECEIVEDWORKREQUEST() then
11:                 $state$.ADD(remaining nodes); break;
12:             remove last element from $V_{used}$
13:             add $UV_i^j$ to end of $V_{used}$
14:             if $N_i$.isGraph then FOUNDMATCH()
15:             for all children $C$ of FATHER($N_i$) do
16:                 if RECEIVEDWORKREQUEST() then
17:                     $state$.ADD(remaining children); break;
18:                 PARALLELMATCH($C, V_{used}$)
19:         remove last element from $V_{used}$
```

Figure 7. Algorithm for resuming a work array

We start by creating the set of used vertices, V_{used} as in `parallelMatch()` (line 2). We then traverse all recursive depths, from the highest to the lowest (line 3), since it would be in that order that they would be computed in the original recursive procedure. Then, we traverse all remaining g-trie nodes (line 5), as it would happen in the inner cycle of `parallelMatch()`, in order to simulate the continuation of the cycle that we stopped. We then follow with all remaining unexplored nodes of that depth (line 9), proceed by updating V_{used} accordingly (line 12 and 13), and then traverse all possible g-trie nodes from that search position (lines 15). In all of these cases, continuation of computation itself is done by calling the original `parallelMatch()` procedure (lines 8 and 18). As in the original procedure, if the computation has to stop, we update the respective work array (lines 6, 7, 16 and 17).

The whole `resumeWork()` is done in order to simulate what would happen if the original recursive procedure kept computing. In fact, if we would artificially stop `parallelMatch()`, and then resume work with `resumeWork()`, we would obtain the exact same results with almost no performance loss. This is due to our efficient array work structure. We experimented with other ways of saving state, like keeping a list of all work units, but that list size would not be really bounded and there would be much redundant information, like common subsets of V_{used}. By using our work array, we minimize the information needed to continue, restricting it to the bare essencial, and we have a strictly bounded way of keeping state. In fact, the maximum theoretical size of the work array is $O(max\_depth\_gtrie \times V(G))$, since we can only go as far as the maximum g-trie depth, and each depth always has at maximum all nodes as unexplored. In practice, the work array will be kept, with very high probability, much lower than this maximum, since all the constraints will reduce the possible candidates.

The fact that our array is small sized is also beneficial because it means we can easily communicate the array to another processor that issued a request for work. The idea is that, upon receiving such a request, the current processor partitions its work array in two valid pieces, continues to compute with one of them and dispatches the other for the requesting processor. This partitioning is done by the procedure `divideWork()`, claled in line 6 of the `parallelCount()` algorithm (Figure 3). In order to maintain the computation balanced, it is crucial that a processor divides his work array as equally as possible. To achieve this goal, we try to equally divide all unexplored vertices, in a round-robin fashion, in the following way:

- **1st Half** maintains currently explored nodes and vertices; gets even numbered unexplored vertices.
- **2nd Half** gets odd numbered unexplored vertices.

This configures what we can call a *diagonal* work split, meaning that we traverse diagonally our tree search space. This always provides two equally sized halfs in terms of the number of unexplored vertices. For a balanced tree this would provide equal work shares. However, the tree may be highly unbalanced, but nevertheless this is our best estimate. We also use a splitting threshold T_{split}, a way of knowing when not to divide our work array. If it is in fact too small, we may spend more time preparing and sending the work-unit than really just computing it. We based our threshold on the distance to the g-trie leaf node: as we get closer, our work-unit will take less time. So, if all remaining unexplored vertices are closer to a leaf than T_{split}, we do not divide work and instead send a “no work available” message to the requesting processor. This threshold could even be dynamically adapted as we discover how much time an average work-unit is taking. In practice, in our particular environment, a constant value was enough (see Section IV).

For the initial work share division (`getInitialWork()`), there are several possibilities. One option would be to put all work on a single processor and start all other processors with an empty share of work. This would have the undesirable side effect of having all processors but one communicating with the root processor for an initial assignment of work. Conscientiously, we opted to equally divide all the initial graph vertices among processors. Thus, instead of being able to start a subgraph in all possible graph nodes, a processor now has its own possible subset for the first node of any subgraph found. We use a round-robin scheme for the division. This corresponds to changing line 10 of algorithm 2 in order to have V_{cand} initiated only with the correspondent graph nodes. So, for example, if we have a graph with 1,000 nodes and we are using 10 processors, processor #1 will start to use as candidate nodes $1, 11, 21, \ldots$, processor #2 will have nodes $2, 12, 22, \ldots$ and so on. If there are less graph nodes than processors, than some processors will not have feasible candidates and will quickly ask for work. It is very hard to have a perfect static division of the work in the beginning since the search tree is highly irregular, but this approach has the potential to give an initial fair division of work that will afterward be adaptively and dynamically balanced.

The frequency by which processors check for incoming messages (`checkMessagesThreshold()`) is important and must be adequately parametrized. If we check too often, we may spend unnecessary time checking while we could instead be computing. If we check to seldom, the receiver may be stuck too long waiting for new work. At the beginning of the computation, there are scarce requests, since every processor is busy. As time goes by, requests become more often, since the search space gets smaller and new work is needed more often. In order to cope with that, we experimented to have an adaptive and dynamic threshold. We started with an initial threshold T_{check} and each time we check for messages and don’t have any, we augment (or multiply) the threshold a little. If we do have a request message, than, by opposition, we diminish it. In practice, in our particular environment, a constant value was enough, and more than that, the threshold could be small since the probing mechanism provided by `MPI_Probe()` was very fast. See Section IV for more details on this.

The receiver-initiated work request (`askForMoreWork()`) is done by sending a non-blocking message to another processor. Several options exist to determine which processors should we address for work, for example, always prefer the last asked processor, or pick processors on a round-robin fashion, or maintain a preferred group of processors. However, since work arrays are small, with no data locality to be exploited, and since the search tree is highly unbalanced, we opted to always choose a random processor. Any other option would possibly have worst case scenarios with a really bad execution time behavior. Sanders [28] gives a more analytical explanation of why random polling works well in practice.

IV. Results

Termination detection (`notFinished()`) is implemented by broadcasting a termination message whenever a processor discovers that no more work is available, after having asked all other processors for work. We could use a more refined termination mechanism, but for a moderate

Table I
THE SET OF FIVE DIFFERENT REPRESENTATIVE REAL NETWORKS USED ON PARALLEL PERFORMANCE TESTING.

Network	$\|V(G)\|$	$\|E(G)\|$	$\frac{\|E(G)\|}{\|V(G)\|}$	Directed	Description	Source
`baywet`	128	2106	16.45	Yes	Food Web of Florida Bay (Wet Season)	[21], [22]
`neural`	297	2148	7.23	Yes	Neural network of *C. Elegans*	[23], [24]
`netsc`	1589	2742	16.21	No	Coauthorships of scientists working on network theory and experiment	[24], [25]
`yeast`	2361	7182	3.04	No	Protein-protein interaction network in budding yeast	[22], [26]
`foldoc`	13356	91471	6.85	Yes	Relationships between terms in computing dictionary	[22], [27]

number of processors (in our environment, a maximum of 192) this is efficient.

After the termination, the frequency counters of all subgraphs are distributed and we need to aggregate the results in a single processor (`aggregateResults()`). Since all processors know beforehand the associated g-trie, all they need to do is share a vector of frequencies, where the vector position uniquely identifies to which subgraph it is referring to. We use native message passing sum reducing capabilities in order to efficiently collect the results.

All experimental results were obtained on a dedicated cluster with 12 SuperMicro Twinview Servers for a total of 24 nodes. Each node has 2 quad core Xeon 5335 processors and 12 GB of RAM, totaling 192 cores, 288 GB of RAM, and 3.8TB of disk space, using Infiniband interconnect. The code was developed in C++ and compiled with gcc 4.1.2. For message passing we used OpenMPI 1.2.7. All measured times are *wall clock times* meaning real time from the start to the end of all processes.

We tested our algorithm in five different representative real complex networks with varied topological properties. Table I summarizes their features.

A. Parallel Performance

In order to study the scalability and performance of our parallel algorithm, we applied it to all five networks. Since the problem we are solving is to count the frequency of a set of subgraphs S_G in another larger graph, we also need to choose S_G. There are many possibilities for this, depending on what we are really trying to discover and it is impossible to show all possible sets. Since g-tries are an efficient and specialized data-structure, there are cases were the results are computed sequentially in seconds, even when using previous methods these would take a considerable amount of time. The main aim of this section is to show that our algorithm is parallelizable and scalable, and therefore we will show possible use cases where the sequential execution time is big enough to justify parallelization to 128 processors. Note that in reality, for someone studying the network, there is always the need for going to cases where the computation takes much time, since increasing the set and size of subgraphs will lead to a better characterization of the network.

We separately show the results for directed and undirected g-tries. In what regards the performance for undirected subgraphs, we apply g-tries for counting all possible subgraphs of a determined size. This corresponds to what is called a *subgraph census*, a network characterization valuable by itself [5]. More than that, by having all possible subgraphs, we show the flexibility and general applicability of our parallelization scheme. We ran the sequential version of our algorithm for increasing subgraph sizes until the time it takes to compute is reasonable for parallel execution on up to 128 processors. We use one hour as the threshold. We also provide the average growth ratio of the execution time whenever the subgraph size is increased by one (shown in table field `average growth`) as it gives us a strong indication of a suitable problem size for each network. We transformed the directed networks in undirected ones by making all edges bidirectional (these networks are identified by the suffix `-un`), so that our testing range is wider in variety. Note that this transformation still provides meaningful results, because an undirected subgraph in a direct network still evaluates connectedness.

Regarding directed subgraphs, we only used the directed networks. Computing a census, as before, by applying a g-trie with all possible subgraphs of a determined size, is not feasible since the number of different subgraphs is much bigger (for example, there are 880,471,142 different classes of isomorphic directed subgraphs of size 7 and only 853 undirected ones of the same size). We still wanted to show a use case where the computation would have meaning, so we use as a base set of subgraphs a random sample of subgraphs from the network. So, instead of having every possible subgraph of a determined size, we have a subset of that that is meaningful for that particular network. We sampled 0.1% of all subgraphs of a determined size by using the `ESU_RAND` method described in [10], computed the set of different classes of isomorphic subgraphs and then started our algorithm with that as our input set of subgraphs to count. Note that even 0.1% of all subgraph occurrences will result in a large set of different classe and completely counting them after will result in a very high percentage of all occurrences, because frequent subgraphs will be in the input set. As before, we ran the sequential version for increasing sizes until the computation took more than one hour and took note of the average growth.

Tables II and III detail the results for the above mentioned methodology, with focus on the absolute speedup obtained. The sequential time only shows the time spent in counting.

Table II
DETAILED ALGORITHM PARALLEL BEHAVIOR FOR UNDIRECTED NETWORKS.

Network	Subgraph size	Sequential time	Average growth	#Subgraphs in g-trie	#CPUs: speedup				
					8	16	32	64	128
`baywet-un`	7	38233.89s	44.9±8.8	853	**7.94**	**15.87**	**31.73**	**63.76**	**127.03**
`neural-un`	7	843212.02s	55.8±15.2	853	**7.90**	**15.87**	**31.43**	**61.69**	**122.78**
`netsc`	9	6141.13s	11.6±0.7	261080	**7.90**	**14.76**	**31.40**	**62.40**	**124.02**
`yeast`	7	187671.34s	31.6±6.2	853	**7.91**	**15.90**	**31.41**	**63.43**	**124.36**
`foldoc-un`	5	5949.54s	307.8±272.2	21	**7.87**	**15.70**	**30.28**	**61.63**	**121.36**

Table III
DETAILED ALGORITHM PARALLEL BEHAVIOR FOR DIRECTED NETWORKS.

Network	Subgraph size	Sequential time	Average growth	#Subgraphs in g-trie	#CPUs: speedup				
					8	16	32	64	128
`baywet`	7	63675.32s	51.1±10.6	288119	**7.86**	**15.72**	**31.37**	**61.18**	**124.80**
`neural`	7	28591.07s	42.5±11.9	347120	**7.91**	**15.53**	**31.93**	**61.10**	**123.04**
`foldoc`	5	13061.18	97.3±45.8	2702	**7.83**	**15.01**	**30.87**	**62.28**	**125.02**

The generation of the g-trie could also be parallelized, but it is always a minor fraction of the cost of the counting function. In fact, the g-trie generation was always less than 5% of the sequential time for the subsets represented in the table. Note that in some cases we could even have that particular g-trie already pre-computed and re-used. All times measured are wall clock times from the beginning to the end of the computation. We also took note of how many different classes of subgraphs we were looking for (field `#Subgraphs in g-trie`).

In the algorithm description we mentioned two parameters that act as thresholds. T_{split} tells where to stop dividing work, and it is measured in distance to a leaf node. T_{check} controls the frequency at which we poll for incoming messages, and it can be measured in terms of number of work units computed or time spent. We empirically tested several values for these parameters and discovered that, for our particular environment, we could identify constant values that consistently produced good results. We chose $T_{split} = 2$, so, whenever all remaining unexplored vertices are within a distance of 2 to a g-trie leaf, we stop diving work. For T_{check}, we noted that using as a unit the time spent was not feasible, because the threshold condition is tested very often, on every recursive call, and the most efficient language primitive for time check would still degrade considerable the algorithm performance if used that often. We opted for using $T_{check} = 100000$ nodes, that would roughly correspond to 0.1s of computation for each message poll. Such a small value is feasible because the MPI implementation available had a very efficient poll primitive (`MPI_Probe`). We are aware that these values can be data and system dependent for optimal performance, and that ideally these should be dynamically computed by our own algorithm, and that is indeed in our future plans. But it remains that they add adaptability to the algorithm, instead of detracting it from being efficient on other cases.

The obtained results clearly show that our algorithm is scalable up to 128 processors and obtains an almost perfect linear speedup for all tested cases. The main bottleneck of the computation is the counting itself, with the final aggregation of results being almost negligible. The time spent in the final aggregation always corresponded to less than 3%.

If we have a more detailed look at how the parallel computation and communication is being managed, we can observe that the behavior is as expected.

Finally, notice the average growth of the time needed for the computations. The standard deviation (except for `foldoc`) is relatively stable and thus we can predict the time needed for increasing the size of subgraphs. Since we obtained almost linear speedups, we could see that by using the 128 processors we would be able to calculate in the same time larger subgraphs. We note that even increasing the size by one can give new meaningful practical results, since a whole new characterization or pattern can be discovered.

B. *Example Application*

We will now illustrate the potential of our parallel algorithm on a real practical application: finding network motifs [6]. These are subgraphs that are overrepresented in the network. Network motifs have a transverse applicability, and there exist already dozens of published papers with a main focus on their discovery in networks from domains so varied as biology (like with protein-protein interaction [29] or transcriptional regulatory networks [30]), software engineering [31], social networks [32] or even electronic circuits [33].

Present methods calculate this overrepresentation by generating an ensemble of similar randomized networks and computing the frequency of the subgraphs both in the original and the random networks. For a typical application, a user generates something like 1,000 random networks [6].

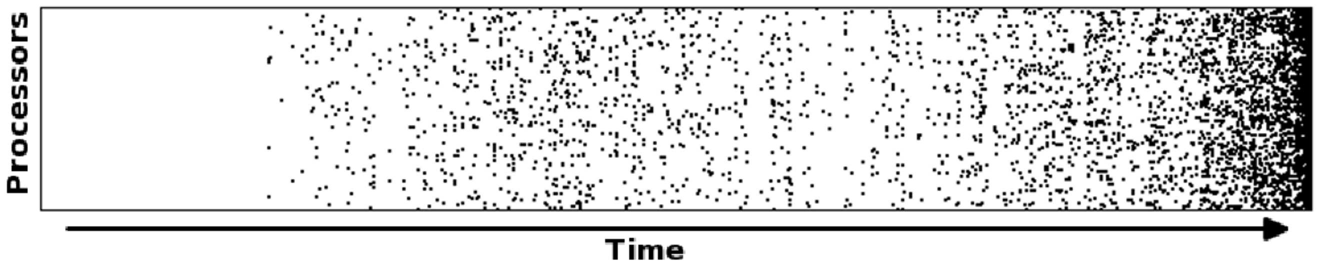

Figure 8. Communication between processors when counting all 7-subgraphs in `yeast` network. White means the processor is busy counting subgraphs, black indicates a processor is asking for work or sharing work.

In order to find the needed frequencies the most customary approach is to compute a subgraph census by enumerating all subgraphs and then computing isomorphisms. As said, to our best knowledge, the ESU algorithm is generally speaking the most efficient algorithm for this task and it is the basis of the well known motif finding tool FANMOD [34]. Another possible approach is to compute the frequency of each subgraph individually, with Grochow and Kellis [11] providing an efficient way of doing that.

If we use our algorithm, we can substantially speedup motifs computation. We can do this by first computing a census of subgraphs of a determined size using ESU [10]. Then we construct a g-trie holding all subgraphs found on the original network (or a subset based on a threshold minimum frequency) and we apply our counting algorithm to all generated random networks, one by one.

Compared to ESU or Grochow and Kellis, sequential g-tries already present a substantial speedup that can be has high as 100$\times$ faster [12] on some networks. If we then add the almost linear speedup obtained by the parallel version of g-tries, we can start to see the potential of this approach, up to more than 1000$\times$ faster than the previoulsy best sequential algorithm, when using 128 processors.

For an empirically verified example, we experimented this approach with `netsc` network. We have an implementation of the ESU and Grochow and Kellis algorithms, comparable in performance with the author's implementations, that uses the same basic code framework as our g-trie implementation. This allows us to isolate execution time differences in the algorithms themselves. We were able to consistently achieve speedups higher than 2000 in relation to the initial sequential network-centric approach, and higher than 500 in relation to the subgraphic-centric approach.

We note that for example ESU allows one to use subgraph sampling in order drastically reduce execution time. However, this comes at the cost of accuracy in the results. It remains that for a complete perfect computation, our approach is a feasible option and we also plan to incorporate sampling in our algorithm in the near future, allowing to gain even more speed, trading it for the loss of some accuracy.

V. Conclusion

In this paper we presented a novel parallel algorithm to count subgraphs. We used as a basis the g-trie data structure, an efficient specialized multiway tree akin to a prefix tree that uses common topologies in subgraphs in order to prune the search three. We created an efficient search state representation, with which we areable to stop, divide and resume the computation. This allowed us to extend and adapt the original g-trie matching algorithm and devise a receiver-initiated scheme in order to obtain dynamic load balancing. We used random polling for work distribution and we collectively aggregate the results in the end. Our algorithm also exhibits extra flexibility and adaptability made possible by the use of throttling parameters that help to dynamically control its behavior.

We tested our algorithm for a set of different representative networks and we achieved an almost linear speedup up to 128 processors in all networks. We have shown that this can allow not only faster response times, but also subgraphs limits previously unfeasible. We have also shown its potential on a multidisciplinary application: finding network motifs. The set of possible uses of the parallel algorithm is however even broader in range.

For future work, we plan to automatically and dinamically compute the threshold parameters T_{split} and T_{check}. We are also exploring the notion of sampling in g-tries in order to give the user the option of trading accuracy for better performance. We will parallelize the network motif computation as whole, in order to avoid having to synchronize processors between each random network. Finally, we want to really put to use our algorithm and apply it on real networks in order to find previously unknown characteristics, like really large network motifs.

Acknowledgments

We thank Enrico Pontelli for the use of Inter Cluster in the New Mexico State University. Pedro Ribeiro is funded by an FCT Research Grant (SFRH/BD/19753/2004). This work was also partially supported by project CALLAS of the FCT (contract PTDC/EIA/71462/2006).

References

[1] R. Albert and A. L. Barabasi, "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, no. 1, 2002.

[2] L. da F. Costa, O. N. Oliveira Jr, G. Travieso, F. A. Rodrigues, P. R. V. Boas, L. Antiqueira, M. P. Viana, and L. E. C. da Rocha, "Analyzing and modeling real-world phenomena with complex networks: A survey of applications," *ArXiv e-prints*, vol. 0711, no. 3199, 2007.

[3] L. da F. Costa, F. A. Rodrigues, G. Travieso, and P. R. V. Boas, "Characterization of complex networks: A survey of measurements," *Advances In Physics*, vol. 56, p. 167, 2007.

[4] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *ACM SIGMOD Conference*, 2004, pp. 335–346.

[5] I. Bordino, D. Donato, A. Gionis, and S. Leonardi, "Mining large networks with subgraph counting," in *8th IEEE International Conference on Data Mining (ICDM)*, 2008, pp. 737–742.

[6] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks." *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[7] J. Köbler, U. Schöning, and J. Torán, *The graph isomorphism problem: its structural complexity*. Basel, Switzerland: Birkhauser Verlag, 1993.

[8] T. Kloks, D. Kratsch, and H. Müller, "Finding and counting small induced subgraphs efficiently," *Information Processing Letters*, vol. 74, no. 3-4, pp. 115–121, 2000.

[9] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," *Data Mining and Knowledge Discovery*, vol. 11, no. 3, pp. 243–271, 2005.

[10] S. Wernicke, "Efficient detection of network motifs," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 3, no. 4, pp. 347–359, 2006.

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

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

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract)," *ACM SIGMETRICS - Performance Evaluation Review*, vol. 13, no. 2, pp. 1–3, 1985.

[14] F. Schreiber and H. Schwobbermeyer, "Towards motif detection in networks: Frequency concepts and flexible search," in *Proceedings of the International Workshop on Network Tools and Applications in Biology (NETTAB04*, 2004, pp. 91–102.

[15] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *IEEE International Conference on Data Mining*, 2001, p. 313.

[16] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *7th International Conference on Database Theory*, 1999, pp. 398–416.

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

[18] T. Wang, J. W. Touchman, W. Zhang, E. B. Suh, and G. Xue, "A parallel algorithm for extracting transcription regulatory network motifs," *IEEE International Symposium on Bioinformatics and Bioengineering*, pp. 193–200, 2005.

[19] P. Ribeiro, F. Silva, and L. Lopes, "Parallel calculation of subgraph census in biological networks," in *1st International Conference on Bioinformatics*, Valencia, Spain, 2010.

[20] M. Schatz, E. Cooper-Balis, and A. Bazinet, "Parallel network motif finding," 2008.

[21] R. Ulanowicz, C. Bondavalli, and M. Egnotovich, "Network analysis of trophic dynamics in south florida ecosystem, fy 97: The florida bay ecosystem," *Technical Report Ref. No. [UMCES] CBL*, pp. 98–123, 1998.

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

[23] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks." *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[24] M. Newman, "Network data sets," `http://www-personal.umich.edu/~mejn/netdata/` , 2010.

[25] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices," *Physical Review E*, vol. 74, no. 3, p. 036104, 2006.

[26] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, G. Li, and R. Chen, "Topological structure analysis of the protein-protein interaction network in budding yeast," *Nucleic Acids Research*, vol. 31, no. 9, pp. 2443–2450, 2003.

[27] E. D. Howe, "The free on-line dictionary of computing," `http://www.foldoc.org/`, 2010.

[28] P. Sanders, "A detailed analysis of random polling dynamic load balancing," in *In International Symposium on Parallel Architectures Algorithms and Networks*. IEEE, 1994, pp. 382–389.

[29] S. Wuchty, Z. Oltvai, and A. Barabasi, "Evolutionary conservation of motif constituents within the yeast protein interaction network," *Nature Genetics*, vol. 35, p. 176, 2003.

[30] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon, "Network motifs in the transcriptional regulation network of escherichia coli," *Nature Genetics*, vol. 31, no. 1, pp. 64–68, 2002.

[31] S. Valverde and R. V. Solé, "Network motifs in computational graphs: A case study in software architecture," *Physical Review E*, vol. 72, no. 2, 2005.

[32] K. Juszczyszyn, P. Kazienko, and K. Musiał, "Local topology of social network based on motif analysis," in *12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part II*, 2008, pp. 97–105.

[33] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, and U. Alon, "Coarse-graining and self-dissimilarity of complex networks." *Physical Review E*, vol. 71, no. 1 Pt 2, 2005.

[34] S. Wernicke and F. Rasche, "Fanmod: a tool for fast network motif detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, 2006.