2014 IEEE International Symposium on Parallel and Distributed Processing with Applications

Parallel Subgraph Counting for Multicore Architectures

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

***Abstract*—Computing the frequency of small subgraphs on a large network is a computationally hard task. This is, however, an important graph mining primitive, with several applications, and here we present a novel multicore parallel algorithm for this task. At the core of our methodology lies a state-of-the-art data structure, the g-trie, which represents a collection of subgraphs and allows for a very efficient sequential search. Our implementation was done using Pthreads and can run on any multicore personal computer. We employ a diagonal work sharing strategy to dynamically and effectively divide work among threads during the execution. We assess the performance of our Pthreads implementation on a set of representative networks from various domains and with diverse topological features. For most networks, we obtain a speedup of over 50 for 64 cores and an almost linear speedup up to 32 cores, showcasing the flexibility and scalability of our algorithm. This paves the way for the usage of such counting algorithms on larger subgraph and network sizes without the obligatory access to a cluster.**

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

I. Introduction

Complex Networks are an ubiquitous representation of systems in many domains [1]. Mining features from these networks is, thus, a very important task with general applicability [2]. One such feature is the number of occurrences of subgraphs. This frequency computation lies at the core of several graph metrics, such as graphlet degree distributions [3] or network motifs [4]. For instance, motifs are over-represented subgraphs, appearing more often than expected. A typical motif discovery algorithm will need to count all subgraphs of a certain size both in the original network and in an ensemble of similar randomized networks [5].

Computing the frequency of subgraphs is, however, a *computationally hard* task, closely related to *subgraph isomorphism*, which is one of the classical NP-complete problems [6]. This means that, as we increase the size of either the subgraphs or the network being analyzed, the execution time increases exponentially. Nevertheless, improving the execution time of subgraph counting can have a broad impact. For example, even increasing by just one node the size of the subgraphs may lead to the discovery of new motifs, providing new insight into a network.

One way to make the subgraph counting algorithms faster is using parallelism. Still, work in this area is very scarce and the vast majority of the existing algorithms are sequential in their nature. We have previous work on the parallelization of subgraph frequency computation, but it was focused on using MPI in distributed environments [7], [8]. Multicore architectures are, however, much more common and readily available to a typical practitioner, with multicores being pervasive even on personal computers.

Our main contribution in this paper is precisely a novel parallel algorithm for subgraph counting geared towards multicores. As a basis, we use our own state-of-art g-trie data structure, which is the core of one of the fastest sequential algorithms for subgraph counting [9]. G-Tries are able to store a collection of graphs, identifying common substructures, and provide an efficient method to search for those graphs as subgraphs of another larger network. This search induces a highly unbalanced search tree with independent tree branches. We use one thread per core and schedule work dynamically based on a diagonal splitting work sharing strategy to try to ensure a fair division of the work. With this technique, we achieve very good performance up to 64 cores and an almost linear speedup up to 32 cores. To the best of our knowledge, this constitutes the fastest multicore algorithm for subgraph counting.

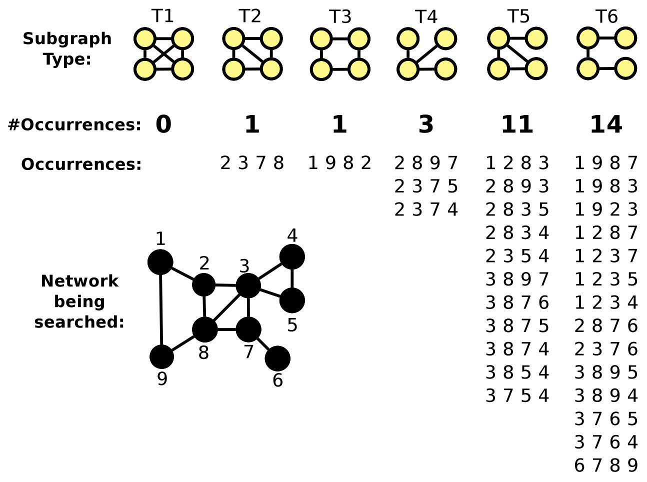
The remainder of this paper is organized as follows. Section II formalizes the problem being tackled and talks about related work. Section III describes the g-trie data structure and its sequential subgraph counting algorithm. Section IV details our parallel approach. Section V shows our experimental results on a series of representative networks. Finally, section VI concludes our paper and gives some possible directions for future work.

II. Subgraph Counting Problem

A. Problem Definition

We start by more formally defining the exact problem we are tackling in this paper:

Definition 1 (General Subgraph Counting Problem): *Given a set of subgraphs S and a graph G, determine the exact count of all induced occurrences of subgraphs of S 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.*



	T1	T2	T3	T4	T5	T6
#Occurrences:	0	1	1	3	11	14
Occurrences:		2 3 7 8	1 9 8 2	2 8 9 7	1 2 8 3	1 9 8 7
				2 3 7 5	2 8 9 3	1 9 8 3
				2 3 7 4	2 8 3 5	1 9 2 3
					2 8 3 4	1 2 8 7
					2 3 5 4	1 2 3 7
					3 8 9 7	1 2 3 5
					3 8 7 6	1 2 3 4
					3 8 7 5	2 8 7 6
					3 8 7 4	2 3 7 6
					3 8 5 4	3 8 9 5
					3 7 5 4	3 8 9 4
						3 7 6 5
						3 7 6 4
						6 7 8 9

Figure 1. An example subgraph counting output, with detailed subgraph occurrences.

Figure 1 gives an example of a subgraph frequency computation, detailing the subgraph occurrences found (these are given as sets of nodes). Note also how we distinguish occurrences: other possible frequency concepts do exist [10], but here we resort to the standard definition.

B. Related Work

Sequential subgraph counting algorithms can be divided into three different conceptual approaches. *Network-centric* methods are based upon the enumeration of all sets of k connected nodes, followed by isomorphism tests to determine the subgraph type of each occurrence. Examples of this strategy include ESU [11], Kavosh [12] and FaSE [13]. By contrast, *subgraph-centric* methods, such as the one by Grochow and Kellis [14], only search for one subgraph type at a time, individually computing their frequency. G-Tries provide a *set-centric* approach, standing conceptually in the middle [9]. They allow the search of a customized set of subgraphs: not necessarily all possible subgraphs of a certain size (as network-centric methods) but also not only one subgraph at a time (as subgraph-centric methods). These algorithms provide exact results, and here we will also concentrate on exact frequency computation, but we should note that there exist some sampling alternatives for providing approximate results. Some examples are Rand-ESU [11], Randomized g-tries [15] and GUISE [16].

Regarding parallel approaches, the available work is scarcer. We provided a distributed memory parallel approach for both ESU [7] and g-tries [8], using MPI for communication. Our work here differs because we instead aim for a shared memory environment with multiple cores. A different parallel approach is the one by Wang et al. [17], which employs a static pre-division of work and limits the analysis to a single network and a fixed number of cores (32). In our work, we use dynamic load balancing and do a more thorough study of the scalability of our approach. A subgraph-centric parallel algorithm using map-reduce was developed by Afrati et al. [18], where they enumerate only one individual subgraph at a time. By contrast, we use a g-trie based set-centric approach and aim for a different target platform (multicores). For more specific types of subgraphs there are other paralell algorithms such as Sahad [19] (an hadoop subgraph-centric method for tree subgraphs), Fascia [20] (a multicore subgraph-centric method for approximate count of non-induced tree-like subgraphs) or ParSE [21] (approximate count for subgraphs that can be partitioned in two by a cut-edge), but our work stands apart by aiming at a completely general set of subgraphs.

III. Sequential G-Trie Algorithm

A. The G-Trie Data Structure

A g-trie is similar in concept to a prefix tree. However, instead of storing strings and identifying common prefixes, it stores subgraphs and identifies common subtopologies. Like a classical string trie, it is a multiway tree, and each tree node contains information about a single subgraph vertex and its connections to the vertices stored in ancestor tree nodes. Descendants of a tree node share a common topology with a path from the root to a node defining a single subgraph. Figure 2 gives an example of a g-trie with the 6 undirected subgraphs previously mentioned stored in its leafs. The same concept can be easily applied to directed subgraphs by also storing the direction of each connection.

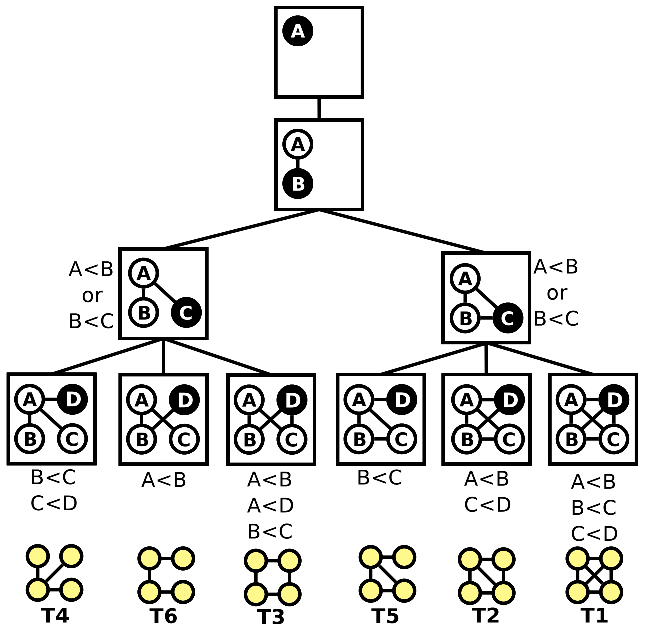


Figure 2. A g-trie representing a set of 6 undirected subgraphs. Each g-trie node adds a new vertex (in black) to the already existing ones in the ancestor nodes (white vertices). Clauses of the form $X < Y$ indicate symmetry breaking conditions.

In order to obtain a unique g-trie representation for a certain subgraph collection, we employ a customized canonical form that tries to ensure that the g-trie is as compact as possible, that is, that we identify as many common subtopologies as possible. This capability is the main strength of a g-trie, not only because we compress the information (avoiding redundant storage), but also because, when we are matching a specific node in the g-trie, we are, at the same time, matching all possible descendant subgraphs stored in the g-trie. In order to avoid symmetries in the stored graphs, g-tries also keep symmetry breaking conditions of the form $X < Y$, indicating that the vertex in position X should have a graph index smaller than the vertex in position Y. Given the space constraints, we refer the reader to [9] for a detailed explanation of how a g-trie can be created. Nevertheless, in the following section we will explain how it can be used to compute the frequency of subgraphs, so that afterwards we can explain how we parallelize the process.

B. Subgraph Counting with a G-Trie

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 graphs. The algorithm depicted in Figure 3 details how we can use g-tries for counting subgraphs sequentially.

```
 1: procedure COUNTALL($T, G$)
 2:     for all vertex $v$ of $G$ do
 3:         for all children $c$ 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 children $c$ 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 $V_{cand}$
```

Figure 3. Algorithm for computing the frequency of subgraphs of g-trie T in graph G.

The core idea of the algorithm is to search for a set of vertices (V_{used}) that match to a path in the g-trie, thus corresponding to an occurrence of the subgraph represented by that path. We use the information stored in the g-trie to heavily constrain the search. In the beginning, all vertices are possible candidates for the initial g-trie root node (lines 2 to 4). Then, we find the set of vertices that fully match with the current g-trie node (line 6) and we traverse that set. If we are at a leaf, we have found an occurrence and increment the respective frequency (line 9). If not, we continue recursively to the other possible g-trie descendants. Function `matchingVertices()` gives some detail on how we efficiently find matches for the current g-trie node. Essentially, from the current partial match, we look for the vertex that is both connected, in the current g-trie node, to the vertex being added and, at the same time, has the smallest number of neighbors in the network, which are the potential candidates for that position (lines 14 and 15). From those vertices, we take the ones that have the exact set of needed connections with the already matched vertices and respect the symmetry breaking conditions stored in the g-trie node (lines 16 to 18).

For the sake of illustration, we will now exemplify how one occurrence is found. We use the notation *(X, k)* to denote that vertex k is matched to X in the g-trie node. Consider Figures 1 and 2 and take for instance the occurrence $\{2, 3, 7, 6\}$ of type $T6$ subgraph. Looking at the respective g-trie leaf, we can see that the only path leading to this occurrence will be $(A, 3) \rightarrow (B, 7) \rightarrow (C, 2) \rightarrow (D, 6)$. A path like $(A, 2) \rightarrow (B, 3) \rightarrow (C, 7) \rightarrow (D, 6)$ could not happen because when adding $(C, 7)$ there would be no matching g-trie node regarding the connections. A path like $(A, 7) \rightarrow (B, 3) \rightarrow (C, 6) \rightarrow (D, 2)$ could not happen either because, even if that would correspond to valid connections, it would break symmetry conditions. In particular, $T6$ imposes the condition $A < B$ and, in this case, 7 is not smaller than 3. These two simple mechanisms (verifying connections and symmetry conditions) form the basis of how a g-trie is able to highly constrain and limit the candidates it is searching and, at the same time, guarantee that each occurrence is found only once.

IV. Parallel G-Trie Algorithm

One of the most important aspects of our sequential algorithm is that it originates completely independent search tree branches. In fact, each call to `count` (T, V_{used}) produces one different branch, and knowing the gtrie node T and the already matched vertices V_{used} is enough for continuing the search from that point. Each of these calls can thus be thought of as a *work unit* and, when designing our parallel algorithm, we aimed to provide a balanced division of work units per resource during execution time.

As we can see in Figure 3, each vertex in the input graph G is given as a candidate for the root node (line 2). A naive approach would be to simply divide these initial work units among the available computing resources. The problem with this static strategy is that the generated search tree is highly irregular and unbalanced. A few of the vertices may take most of the computing time, leading to some resources being busy processing them for a long time while others were idle. To achieve a scalable approach, an efficient dynamic sharing mechanism, that redistributes work during execution time, is required.

Another important factor in the sequential algorithm's performance is that there is no explicit queue of unprocessed work units. Instead, the recursive stack implicitly stores the work tree, with the two cycles between vertices and nodes (lines 7 and 11) being responsible for generating new work units that are recursively processed (line 12). In our parallel approach we keep this crucial feature of the algorithm and do not artificially introduce explicit queues during the normal execution of the algorithm. These queues would introduce a serious overhead both on the execution time and on the needed memory, significantly deteriorating the sequential algorithm's performance. Our goal is, therefore, to scale up our original efficient algorithm, providing the best possible overall running time.

Since we want the end users to take advantage of their personal multicore machines, our target is a shared memory architecture. For that purpose we chose Pthreads, due to its portability and flexibility. [1]

A. Overall View

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

B. Parallel Subgraph Frequency Counting

Figure 4 depicts our parallel counting algorithm. All threads start by executing `parallelCountAll()` with an initially empty work tree W (line 2). The first vertex that a thread computes is that of position $thread_{id}$ (lines 3 and 5). At each step, the thread computes the vertex $thread_{id}$ positions after the previous one (line 13). Every vertex is used as a candidate for the g-trie root node by some thread (lines 11 and 12). This division gives approximately $|V(G)|/num_{threads}$ vertices for each thread to initially explore. We do this in a round-robin fashion because it generally provides a more equitable initial division than

[1]Our implementation, along with test data, can be consulted on the following URL: `http://www.dcc.fc.up.pt/gtries/`.

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

Figure 4. Parallel algorithm for computing the frequency of subgraphs of g-trie T in graph G.

simply allocating continuous intervals to each thread, due to the way we use the symmetry breaking conditions. Our intuition was verified empirically by observing that the threads would ask for work sooner if continuous intervals were used. When a thread Q receives a work request from P (line 6) it needs to stop its computation, save what it still had left to do (line 7), divide the work tree (line 8), give P some work (line 9) and resume the remaining work (line 10). On the other hand, if a thread finishes its initially assigned work, it issues a work request to get new work (line 14).

`parallelCount()` remains almost the same as the sequential version, except for now attending work requests and storing subgraph frequencies differently. If the thread receives a work request while computing matches, it first adds them to the work tree W and then stops the current execution (lines 18 to 20) to compute the current state and build the work tree. In the sequential version we simply needed to increase the frequency of a certain subgraph in the g-trie structure. As for the parallel version, multiple threads may be computing frequencies for the same subgraph, using different vertices from the input graph, and so they need to coordinate their frequency storing. Initially, we kept in each g-trie node a shared array $Fr[1..num_{threads}]$ where the threads would update the array at the position of their $thread_{id}$. In the end, the global frequencies would be obtained by summing the values in the array. This resulted in significant false sharing due to too many threads updating those arrays simultaneously, and became a severe bottleneck.

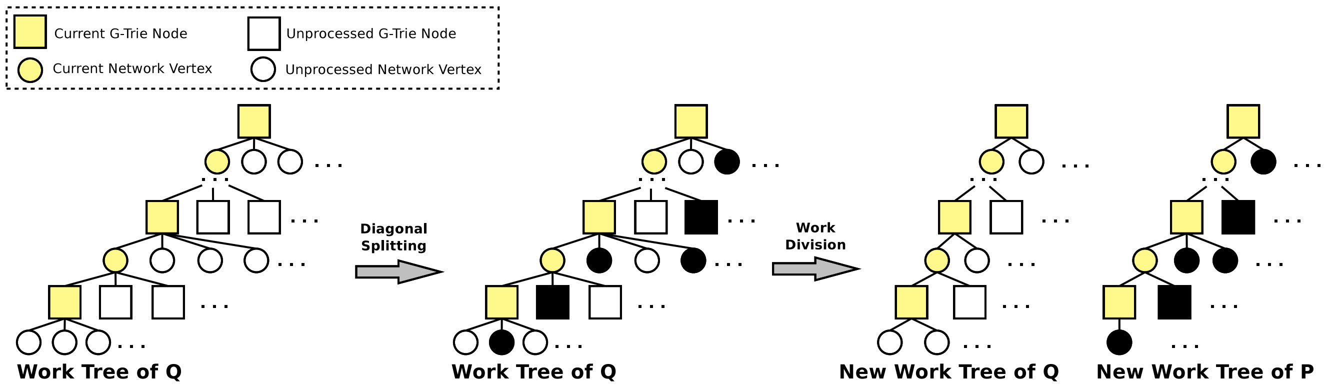


Figure 5. The constructed work tree of a thread Q and its division by diagonal splitting when a work request is received from thread P.

Our solution was to create thread private arrays indexing g-trie nodes, i.e. $Fr[1..num_{gtrieNodes}]$, which impacted very favorably our efficiency. In our testing phase with a 24 cores machine, we had cases with speedups below 5 that, only with this change, went to a speedup of over 22, thus converting a modest into an almost linear speedup.

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

C. Work Request

A work request is performed when some thread P has completed its assigned work. Since there is no efficient way of predicting exactly how much computation each active thread still has in its work tree, it asks a random thread Q for more work. Note that this kind of random polling has been established as an efficient heuristic for dynamic load balancing [22]. If Q sends some unprocessed work, then P executes the `resumeWork()` procedure. If Q does not have any work to share, P proceeds by asking another random thread. The computation is over when all threads are requesting work and thus no more work units remain to be processed.

D. Work Sharing

When a thread Q receives a work request it builds a work tree representing its current recursive state. In Figure 5 we show a resulting work tree and its division with a caller thread P. The yellow colored circles constitute V_{used} and the yellow colored squares form the g-trie path up to the current level. The other nodes and vertices are still left to be explored and are split in a round-robin fashion. This division results in two work trees with approximately the same number of work units. This does not, however, imply that the two halves are of the same computational dimension given the irregularity of the search tree they will induce, but, nevertheless, they constitute our best guess of a fair division across all levels.

As said before, we only build an explicit work tree when a work request is received. In that situation, a thread saves the current and the other unexplored vertices for the current node and moves up in the recursive tree. This process is repeated up to the top level, effectively populating the work tree with the unprocessed work units, i.e., the unexplored g-trie nodes and network vertices. This is a very fast operation and it is done by stopping the execution of the recursive `parallelCount()` calls and adding the work to the work tree (line 19 in Figure 4) until we get to `parallelCountAll()` and add the remaining nodes and vertices of the top level (line 7). We also store the current g-trie path and network vertices (V_{used}).

E. Work Resuming

After the threads have shared work, they resume it and proceed with the computation. The work tree W is traversed in a bottom-up fashion (lines 2 to 5) and the vertices of each level are computed (line 6). If the thread receives a work request, work sharing is performed (line 7). There is no call to `addWork()` since the work

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

Figure 6. Algorithm for resuming work after sharing is performed.

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

Network	$\|V(G)\|$	$\|E(G)\|$	$\frac{\|E(G)\|}{\|V(G)\|}$	Directed	Description	Ref.	Source
`polblogs`	1,491	19,022	12.76	Yes	Network of hyperlinks between weblogs on US politics	[23]	Newman[1]
`netsc`	1,589	2,742	1.73	No	Coauthorships of scientists working on network experiments	[24]	Newman[1]
`facebook`	4,039	88,234	21.85	No	Friend circles from Facebook	[25]	SNAP[2]
`routes`	6,474	12,572	1.94	No	Traffic flows between routers	[26]	SNAP[2]
`company`	8,497	6,724	0.79	Yes	Ownership of media and telecommunication companies	[27]	Pajek[3]
`blogcat`	10,312	333,983	32.39	No	Friendship and group membership networks from BlogCat	[28]	ASU[4]
`enron`	36,692	367,662	10.02	Yes	Communication network of around half a million emails	[29]	SNAP[2]

is already on the work structure: either it was unfinished work already on W or it was added there by the recursive `parallelCount()` calls. After work sharing is performed (lines 8 and 9), the thread continues its computation with the new work tree (line 10) and the current execution is discarded (line 11). On the other hand, if it does not have a pending work request, it proceeds to process the vertex. The thread first checks if it has arrived at a desired subgraph (line 12) and increases its frequency in that case (line 13). Otherwise, the thread calls `parallelCount()` with the new vertex added to V_{used} for each children of the g-trie node (lines 15 and 16). When all work is completed it requests work from another thread (line 17).

V. Results

Our experimental results were gathered on a 64-core machine. It consists of four 16-core AMD Opteron 6376 processors at 2.3GHz with a total of 252GB of memory installed. Each 16-core processor is split in two banks of eight cores, each with its own 6MB L3 cache. Each bank is then split into sets of two cores sharing a 2MB L2 and a 64KB L1 instruction cache. A 16KB L1 data cache is dedicated to each core. We disabled the turbo boost functionality because it would give us inconsistent results by having executions with less cores running at an increased clock rate. All code was developed in C++11 and compiled using gcc 4.8.2. We used NPTL 2.18 for Pthread support. To measure real times, we opted for the `gettimeofday()` function.

We tested our algorithm in a few dozens of real-world networks and here we present the results for a representative subset of them. Table I gives a general view of the content and dimension of the chosen seven networks. In order to showcase the general scalability of our algorithm, we chose networks that vary in their field of application, their use of edge direction and their dimension, as can be seen in Table I. At the same time, we need to choose which subgraphs should be searched in the networks. For that purpose we use all possible subgraphs of a given size k, again to highlight general applicability. Note that when we consider directed networks, the number of possible subgraphs of size k increases drastically. For example, for $k = 4$ there are only 6 undirected graphs and 199 directed. One query on a directed network for $k = 4$ would therefore imply counting the occurrences of 199 different types of subgraphs.

The g-trie sequential algorithm takes a few seconds in cases where competing algorithms would take a considerable amount of time [9]. Our purpose here is to explicitly pick very large cases even for g-tries. The sequential time for the examples used range from a couple of minutes to several hours. We chose this approach to show the real importance of our work, since going from a few seconds to tenths of seconds is of minimal practical interest to the user. Searching for larger subgraphs or networks takes longer but can provide new important insight and, from a practitioner point of view, our parallel approach increases the limits of what is feasible to compute on a reasonable amount of time.

As said before, we wanted our parallel strategy with one thread to perform similarly to the original sequential version. Empirically we observed that our parallel implementation with one thread does not produce a high overhead, being on average less than 10% for all the networks we tested (the maximum overhead we obtained was 17%). The overhead lies mostly in threads having to check if they received a work request, with sharing itself having minimal impact. Using code profilers, such as Intel VTune and AMD CodeXL, we verified that sharing took a negligible amount of time (less than 1% of the total time). Henceforth we will use *sequential time* to mean the execution with only one thread and the speedups shown are relative to it.

Our algorithm was evaluated up to 64 cores and here present the results in three tables. In Table II we show the size of the subgraphs and the resulting number of all possible subgraphs of that type and size that will be counted in that network. The sequential time and the obtained speedups for 8, 16, 32 and 64 cores are shown in Tables III and IV. We present two tables containing the speedups with and without compiler optimization (gcc `-O3` flag) because we observed significant differences in the results. This happens due to compiler optimizations usually not being

[1]Mark Newman: `http://www-personal.umich.edu/~mejn/netdata/`
[2]SNAP: `http://snap.stanford.edu/data/`
[3]Pajek: `http://vlado.fmf.uni-lj.si/pub/networks/data/`
[4]ASU: `http://socialcomputing.asu.edu/pages/datasets`

Table II
OVERALL EXECUTION INFORMATION.

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

Table III
RESULTS WITH COMPILER OPTIMIZATIONS.

Sequential time (s)	#Threads: speedup			
	8	16	32	64
91,190.73	**7.87**	**15.69**	**31.31**	**52.96**
466.48	**7.90**	**15.78**	**30.91**	**51.09**
6,043.90	**6.75**	**14.72**	**30.23**	**52.47**
4,936.54	**6.53**	**14.52**	**30.34**	**48.76**
26,955.71	**6.74**	**14.54**	**29.99**	**45.12**
5,410.45	**7.72**	**14.37**	**24.92**	**25.69**
1,038.60	**6.23**	**12.69**	**23.78**	**24.41**

Table IV
RESULTS WITHOUT COMPILER OPTIMIZATIONS.

Sequential time (s)	#Threads: speedup			
	8	16	32	64
222,210.76	**7.91**	**15.78**	**31.38**	**52.11**
2,030.39	**7.91**	**15.74**	**31.36**	**51.65**
17,851.16	**6.78**	**14.67**	**30.31**	**53.84**
20,706.67	**6.80**	**14.67**	**30.53**	**52.44**
94,384.39	**6.69**	**14.61**	**30.17**	**47.09**
15,666.05	**7.88**	**15.40**	**29.60**	**48.69**
2,768.74	**6.42**	**13.69**	**27.43**	**45.59**

designed for parallel programs. For example, some cache optimizations that greatly reduce the sequential time are not as effective when multiple cores are running at the same time. This effect may cause an unfair comparison between sequential and parallel executions. Nevertheless, results from Table III are also positive and users will be more interested in real execution time than speedups, therefore we decided to include both tables for the sake of completeness.

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

We can also observe that as the network size increases, the performance slightly degrades. This is particularly noticeable in the two largest networks, which show the worst behavior. This is mostly due to their large size leading to memory constraints and cache issues. Note, however, that their behavior without compiler optimizations is not significantly worse. Furthermore, here we used an adjacency matrix to represent the network. This gives the best possible algorithmic complexity for verifying if an edge exists but, at the same time, implies a quadratic representation in memory. Other data structures would degrade edge verification performance but also significantly decrease the memory footprint, and thus would contribute to a potentially better shared memory parallel performance. We should also note that we have done previous work in a distributed memory environment on which we obtained near-linear speedup up to 128 processors [8]. In that architecture, each CPU has its own dedicated main memory storing a copy of the graph, which means that the problems related with competing memory and caching are not present.

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

VI. CONCLUSION

In this paper we presented a scalable algorithm to count subgraph frequencies for multicore architectures. We used the g-trie data structure as a basis for parallelization. G-Tries are multiway trees, much like prefix trees, that use common topologies in subgraphs in order to prune the search tree. The sequential version already performed significantly better than competing algorithms, making it a solid base for improvement. We were able to keep the original recursive nature of the counting algorithm and only create a more explicit work tree when needed. To dynamically divide the search tree among the threads, we developed an efficient sharing mechanism that is able to stop, split and resume the execution. Our algorithm was tested in several representative networks from various fields and presented near-linear speedup up to 32 cores and a speedup of over 50 for 64 cores. To the best of our knowledge, our parallel algorithm is the fastest available method for shared memory environments and allows practitioners to take advantage of either their personal multicore machines or more dedicated computing resources. This expands the limits of subgraph counting applicability, allowing an exploration of larger subgraphs in bigger networks.

In the near future, we intend to investigate the potential of mixing the shared and distributed memory approaches since both of them showed promising results. We are also interested in experimenting with GPU programming because powerful GPUs are now commonplace and their manycore architecture may lead us to interesting speedups. We also intend to explore several variations on the g-tries algorithm, like, for instance, using different base graph data-structures or using sampling to obtain approximate results. Finally, to give our work a more practical context, we will use our implementation in real world scenarios. For example, we are in the process of building a large co-authorship network and plan to explore its structure using our algorithm.

Acknowledgments

David Aparício is funded by an FCT Scolarship Grant from the project Sibila (NORTE-07-0124-FEDER-000059) financed by *Fundo Europeu do Desenvolvimento Regional (FEDER)*. Pedro Ribeiro is funded by an FCT Research Grant (SFRH/BD/19753/2004).

References

[1] 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.

[2] 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.

[3] N. Pržulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 23, pp. e177–e183, Jan. 2007.

[4] 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.

[5] P. Ribeiro, F. Silva, and M. Kaiser, "Strategies for network motifs discovery," in *5th IEEE International Conference on e-Science*. Oxford, UK: IEEE CS Press, 2009.

[6] S. A. Cook, "The complexity of theorem-proving procedures," in *3rd annual ACM symposium on Theory of computing*, ser. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158.

[7] P. Ribeiro, F. Silva, and L. Lopes, "Parallel discovery of network motifs," *Journal of Parallel and Distributed Computing*, vol. 72, pp. 144–154, 2012.

[8] ——, "Efficient parallel subgraph counting using g-tries," in *IEEE International Conference on Cluster Computing (Cluster)*. IEEE Computer Society Press, September 2010, pp. 1559–1566.

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

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

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

[12] Z. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad, "Kavosh: a new algorithm for finding network motifs," *BMC bioinformatics*, vol. 10, no. 1, p. 318, 2009.

[13] P. Paredes and P. Ribeiro, "Towards a faster network-centric subgraph census," in *International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2013, pp. 264–271.

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

[15] P. Ribeiro and F. Silva, "Efficient subgraph frequency estimation with g-tries," in *10th International Workshop on Algorithms in Bioinformatics*. Springer, September 2010, pp. 238–249.

[16] M. Bhuiyan, M. Rahman, M. Rahman, and M. A. Hasan, "Guise: Uniform sampling of graphlets for large graph analysis," in *IEEE International Conference on Data Mining*, ser. ICDM, 2012, pp. 91–100.

[17] 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.

[18] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *IEEE 29th International Conference on Data Engineering (ICDE)*. Los Alamitos, CA, USA: IEEE CS, 2013, pp. 62–73.

[19] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. A. Kumar, and M. V. Marathe, "Sahad: Subgraph analysis in massive networks using hadoop," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 390–401, 2012.

[20] G. M. Slota and K. Madduri, "Fast approximate subgraph counting and enumeration," in *42nd International Conference on Parallel Processing (ICPP)*, 2013, pp. 210–219.

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

[22] P. Sanders, "A detailed analysis of random polling dynamic loadbalancing," in *International Symposium on Parallel Architectures Algorithms and Networks*. IEEE, 2002, pp. 382–389.

[23] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 u.s. election: Divided they blog," in *3rd International Workshop on Link Discovery*, ser. LinkKDD '05. New York, NY, USA: ACM, 2005, pp. 36–43.

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

[25] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Advances in Neural Information Processing Systems 25*, P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., 2012, pp. 548–556.

[26] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: ACM, 2005, pp. 177–187.

[27] K. Norlen, G. Lucas, M. Gebbie, and J. Chuang, "EVA: Extraction, Visualization and Analysis of the Telecommunications and Media Ownership Network," in *International Telecommunications Society 14th Biennial Conference (ITS2002), Seoul Korea*, 2002.

[28] L. Tang and H. Liu, "Relational learning via latent social dimensions," in *15th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD09)*, 2009, pp. 817–826.

[29] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.