A Scalable Parallel Approach for Subgraph Census Computation

David Aparicio, Pedro Paredes, Pedro Ribeiro
{daparicio, pparedes, pribeiro}@dcc.fc.up.pt

CRACS & INESC-TEC, Faculdade de Ciencias, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal

Abstract. Counting the occurrences of small subgraphs in large networks is a fundamental graph mining metric with several possible applications. Computing frequencies of those subgraphs is also known as the subgraph census problem, which is a computationally hard task. In this paper we provide a parallel multicore algorithm for this purpose. At its core we use FaSE, an efficient network-centric sequential subgraph census algorithm, which is able to substantially decrease the number of isomorphism tests needed when compared to past approaches. We use one thread per core and employ a dynamic load balancing scheme capable of dealing with the highly unbalanced search tree induced by FaSE and effectively redistributing work during execution. We assessed the scalability of our algorithm on a varied set of representative networks and achieved near linear speedup up to 32 cores while obtaining a high efficiency for the total 64 cores of our machine.

Keywords: Graph Mining, Subgraph Census, Parallelism, Multicores

1 Introduction

Graphs are a flexible and powerful abstraction of many real-life systems. An essential graph mining primitive is to compute the frequency of small subgraphs in large networks. This is known as the *subgraph census* problem, and lies at the core of several graph mining methodologies, such as network motifs discovery [6] or graphlet based metrics [8]. Counting subgraphs is, however, a *computationally hard* task, closely related to *subgraph isomorphism*, a classical NP-Complete problem . This implies that the execution time grows exponentially with the size of the network or the subgraphs being analyzed. Speeding up this computation would have a significant and broad impact, making new size limits computationally feasible, hence leading to a new insight on the networks.

Subgraph census algorithms generally follow one of three different paradigms; *network-centric* algorithms, such as `ESU` [16], compute the frequency of all subgraphs with a certain number of nodes and then verify the type of each subgraph. By contrast, *subgraph-centric* algorithms, such as the one by Grochow and Kellis [3], compute the frequency of only one individual subgraph type at a time. *Set-centric* approaches, such as g-tries as used in [9], are conceptually in the middle and allow the user to compute the frequency of a customized set of subgraphs that can be larger than a single subgraph but at the same time smaller than all possible subgraphs of a certain size.

Here we are mainly concerned with the network-centric approach. In particular, we focus on the `FaSE` algorithm which is one of the most efficient sequential alternatives for this conceptual approach to subgraph census [7]. The main contribution of this paper is a scalable parallel version of `FaSE` geared towards multicore architectures, which are nowadays ubiquitous, even on personal computers, making them an ideal target for end users. Using an efficient dynamic load balancing scheme our parallel algorithm is able to redistribute the work contained in the highly unbalanced search tree produced by `FaSE`. We tested our approach on a series of representative networks, obtaining very promising results, with an almost linear speedup up to 32 cores and high efficiency for 64 cores. Sequential `FaSE` was already one or two orders of magnitude faster than state-of-the-art algorithms and so our parallel version constitutes, to the best of our knowledge, the fastest multicore network-centric algorithm.

The remainder of this paper is organized as follows. Section 2 formalizes the problem and describes related work. Section 3 gives an overview of the sequential `FaSE` algorithm. Section 4 details our parallel approach, while section 5 shows our experimental results. Finally, section 6 sums up the presented work and gives some possible directions for future research.

2 The Subgraph Census Problem

This section details more formally the problem tackled in this paper.

Definition 1 (Subgraph Census Problem) *Given an integer k and a graph G, determine the frequency of all connected induced subgraphs of size k in G. Two occurrences of a subgraph are considered different if they have at least one node that they do not share.*

As previously stated, this metric plays a central role in several graph mining methodologies. For instance, a network motif is defined as a statistically *overrepresented subgraph*, that is, a subgraph that appears more times than what would be expected [6]. In practice, this means that the census must be computed both on the original network and on an ensemble of randomized networks [10].

2.1 Related Work

There are several existing sequential algorithms for the subgraph and classical examples are `ESU` [16] and `Kavosh` [4]. They are conceptually similar, both being network-centric and enumerating all possible subsets of k connected nodes, relying on a third-party algorithm (`nauty`[1]) to identify the associated subgraph

[1] `http://cs.anu.edu.au/~bdm/nauty/`

type. This means that each subgraph occurrence implies an individual isomorphism test. NetMODE augments this approach by considering very small subgraph sizes and either caching isomorphism tests or building fast specialized heuristics for a particular subgraph size. QuateXelero [5] and our own work with FaSE [7] are two very recent algorithms which offer a different improvement by avoiding the need to do one isomorphism test per occurrence. To that end, they both encapsulate the topology of the subgraphs being enumerated on an auxiliary data-structure (a quaternary tree in the case of QuateXelero, and a g-trie in the case of FaSE). Other algorithms are either subgraph-centric, such as the work by Grochow and Kellis [3] or set-centric, such as gtrieScanner [9]. Here we concentrate on the network-centric approach and use FaSE as the basis for our parallel algorithm.

Regarding parallel approaches, there are less alternatives. We provided a distributed memory approach for both ESU [12] and g-tries [11], using MPI. This work stands out because it is aimed at shared memory environments with multiple cores. A shared memory parallelization of the set-centric g-trie methodology was also presented in [2]. This work diverges in its base sequential algorithm and uses a different conceptual approach. Another parallel algorithm is given by Wang et al [15]; however, they employ a static pre-division of work and provide very limited experimental results while our approach dynamically balances load by redistributing work during the computation and perform a more detailed scalability analysis. Afrati et al. [1] provide a parallel map-reduce subgraph-centric approach, from which we differ in both the target platform and the algorithmic methodology. For more specific subgraph types there are other parallel alternatives such as Fascia [14] (a multicore subgraph-centric method for approximate count of non-induced tree-like subgraphs) or Sahad [17] (a Hadoop subgraph-centric method for tree subgraphs), but here we aim towards generality and all possible subgraph types.

3 Sequential FaSE Algorithm

As previously said, FaSE follows a network-centric paradigm. However, contrarily to what previous approaches did, FaSE does not withhold the isomorphism tests until the end of the enumeration. Instead, it partitions the subgraphs into intermediate classes during the enumeration process. The only requisite is that if two subgraphs pertain to the same intermediate class they are isomorphic. Thus, a single isomorphism test per intermediate class is needed, contrasting to previous methods that required one per enumerated subgraph. This results in a major speedup when comparing with past approaches, since the number of intermediate classes will be much smaller than the number of subgraph occurrences, which is corroborated by the experimental results.

In practice the algorithm uses two main concepts: an enumeration process and a tree that stores the information of both the intermediate classes and the subgraphs being enumerated. The enumeration process simply iterates through each subgraph occurrence and can be performed using any existing methods, provided it works by incrementally growing a set of connected vertices that partially represents the current subgraph. Furthermore, a tree is used to encapsulate the topological features of the enumerating subgraphs. It does so by generating a new label, using a generic operation called *LS-Labeling*, which represents the information introduced by each newly added vertex and uses it to describe an edge in a tree. This effectively partitions the set of subgraphs into the mentioned intermediate classes. This entire process is summarized in Algorithm 1.

Algorithm 1 The `FaSE` Algorithm

Input: A graph G, a g-trie T and a subgraph size k
Result: Frequencies of all k-subgraphs of G

```
 1: procedure FaSE($G, T, k$)
 2:     $T \leftarrow \emptyset$
 3:     for all vertex $v$ of $G$ do
 4:         ENUMERATE($\{v\}, \{u \in N(v) : u > v\}, T.root$)
 5:     for all $l$ in $T.leaves()$ do
 6:         $frequency$[CANONICALLABEL($l.Graph$)] += $l.count$
 7: procedure ENUMERATE($V_s, V_{ext}, current$)
 8:     if $|V_s| = k$ then
 9:         $current.count$++
10:     else
11:         for all vertex $v$ in $V_{ext}$ do
12:             $V'_{ext} \leftarrow V_{ext} \cup \{u \in N_{exc}(v, V_s) : u > V_s[0]\}$
13:             $V'_s \leftarrow V_s \cup \{v\}$
14:             $current' \leftarrow current.Child(LSLabel(V_s))$
15:             ENUMERATE($V'_s, V'_{ext}, current'$)
```

3.1 Enumeration

As mentioned above, the enumeration process can be done by any algorithm that grows a set of connected vertices. The reason to enforce so is to allow the creation of a label describing the addition of the vertex and hence partition the subgraphs set. The previously mentioned `ESU` [16] and `Kavosh` [4] algorithms fit this constraint and since they present similar execution time, both would be a good choice to integrate into `FaSE`. In our implementation we opted to use `ESU`, which we will now describe in more detail.

It essentially works by enumerating all size k subgraphs only once. It does so by keeping two ordered sets of vertices: V_s and V_{ext}. The former represents the partial subgraph that is currently being enumerated as a set of connected vertices. The latter represents the set of vertices that can be added to V_s as a valid extension. To begin, it takes each vertex v in the network sets $V_s = \{v\}$ and $V_{ext} = N(v)$, where $N(v)$ are the neighbors of v (lines 3 and 4). Then, one element u of V_{ext} is removed at a time, and a recursive call is made adding u to V_s and each element in $N_{exc}(u, V_s)$ with label greater than $V_s[0]$ to V_{ext} (lines 12 and 13). $N_{exc}(u, V_s)$ are the exclusive neighbors, that is they are the neighbors of u that are not neighbors of V_s. This, along with the condition $u > V_s[0]$, ensure that there is no subgraph enumerated twice. When the size of V_s reaches k it means that V_s constitutes a new occurrence of a size k subgraph (line 8).

3.2 Using a Tree to Encapsulate Isomorphism Information

The enumeration step is wrapped by a data structure that stores information of the subgraphs being enumerated in order to divide them into intermediate classes. The conditions set on the behavior of the enumeration algorithm allow for the use of a tree, as previously described. This tree, which is called a g-trie, represents a different intermediate class in each node. When adding a new vertex to the current subgraph, a new label is generated describing its relation to the previously added vertices. This label will govern the edges in the tree, that is, each edge is represented by a label generated by a vertex addition.

Label generation in each step is done by using a generic process called *LS-Labeling* which deterministically partitions the different subgraphs into isomorphic classes. Additionally, it is required that it runs in polynomial time, as otherwise it would be pointless to use the actual tree since we could simply use the labeling as the isomorphism test. Thus there is a trade off between time spent in creating the label and time spent enumerating and running isomorphism tests on subgraphs. In this paper we use an *adjacency list labeling*, which generates a label corresponding to an ordered list of at most $k-1$ integers where each value i $(0 < i < k)$ is present if there is a connection from the new vertex to the i-th added vertex. More details on this can be found in the original FaSE paper [7].

Figure 1 summarizes the FaSE algorithm. The tree on the left represents the implicit recursion tree ESU creates. Note that it is naturally skewed towards the left. This is an important fact that justifies why, as we will see later, we need to redistribute work in the parallel version of the algorithm. The induced g-trie on the right is a visual representation of the actual g-trie that FaSE creates.

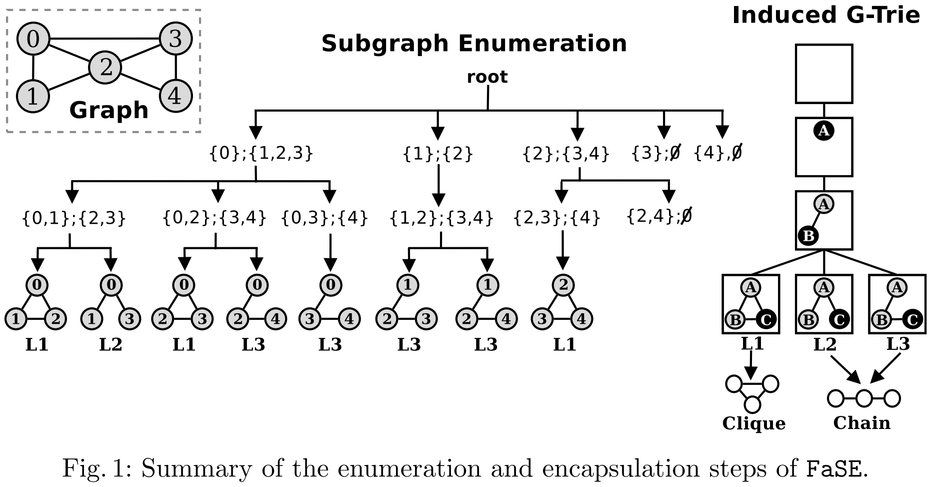

Fig. 1: Summary of the enumeration and encapsulation steps of FaSE.

4 Parallel FaSE Algorithm

A main characteristic of our sequential algorithm is that it generates independent branches. Each V_s and V_{ext} pair can thus be regarded as a *work unit* and, along with the position in the g-trie, are sufficient to resume computation. At the start, V_s corresponds to each single node in the network and V_{ext} to its neighbors with higher index. As we have seen before, this distribution is intrinsically unbalanced since it places bigger restrictions on higher indexed nodes. Furthermore, in the subgraph census problem, a few vertices, such as hubs, may have most of the computing time while others are much lighter in comparison. In our work we developed a strategy to efficiently distribute these work units among the computing resources.

We decided to use one central g-trie, as opposed to one g-trie per thread. While this option leads to contention when accessing the g-trie, it saves memory and removes the redundant work caused by multiple threads creating their own g-trie, with most connections being common for every thread. A major factor for the efficiency of the sequential algorithm is that it does not create a queue of work units, and instead implicitly stores them in the recursive stack. To achieve the best efficiency we kept this characteristic in our parallel approach.

Our target platforms are multicore architectures, given their ubiquity and ease of access for end users. Our implementation was done using Pthreads, which are supported by all major operating systems.

4.1 Overall View

The algorithm starts by dividing the vertices evenly between the threads, with one thread per core. All threads do the enumeration process separately, using their respective V_s and V_{ext}. If a thread arrives at a new type of node it updates the g-trie. All threads see this change and do not need to update the g-trie if the node is found again. When a thread P finishes its initially assigned work units it sends a work request to an active thread Q. Thread Q stops its computation, builds a work tree corresponding to its current state, gives half of the work to P and informs it that it can resume work. Both threads execute their respective portion starting at the bottom of the work tree so that only one V_s is needed for a given point of sharing, exploiting graph sub-topology between g-trie's ancestor and descendant nodes. After the enumeration phase is completed, the resulting leafs are split between the threads and isomorphism tests are performed to assert to which subgraph type each leaf corresponds to. In the end, the subgraph frequencies computed by all threads are aggregated.

4.2 Parallel Subgraph Frequency Counting

Algorithm 2 details our parallel `FaSE` algorithm. The graph G, the g-trie T and the subgraph size k are global variables, while *current* is a pointer to the g-trie location and is local for each thread. Computation starts with an initially empty g-trie (line 2) and work queues (line 3) for every thread. The condition in

Algorithm 2 The Parallel `FaSE` Algorithm

Input: A graph G, a G-Trie T and a subgraph size k
Result: Frequencies of all k-subgraphs of G

```
 1: procedure ParallelFaSE($G, T, k$)
 2:     $T \leftarrow \emptyset$
 3:     $W \leftarrow \emptyset$
 4:     $i, j \leftarrow thread_{id}$
 5:     while $i \leq |V(G)|$ do
 6:         $v \leftarrow V(G)_i$
 7:         if workRequest($P$) then
 8:             $W$.addWork()
 9:             $(W_Q, W_P) \leftarrow$ splitWork(W)
10:             giveWork($W_P$, $P$)
11:             resumeWork($W_Q$)
12:         enumerate($\{v\}, \{u \in N(v) : u > v\}, T.root$)
13:         $i \leftarrow i + num_{threads}$
14:     while $j \leq |T.leaves()|$ do
15:         $l \leftarrow T.leaves()_j$
16:         $frequency$[canonicalLabel($l.Graph$)] += $l.count$
17:         $j \leftarrow j + num_{threads}$
```

line 12 of Algorithm 1, $u > V_s[0]$, makes vertices with a smaller index probably computationally heavier than higher indexed vertices. For that reason, network vertices are split in a round-robin fashion, giving all threads $|V(G)|/num_{threads}$ top vertices to initially explore (lines 4 to 6 and 13). This division is not necessarily balanced but finding the best possible division is as computationally heavy as the census itself. If a thread does not receive a work request it does the enumeration process starting at each of its assigned vertices (line 12). The `enumerate()` procedure is very similar to the sequential version but with V_s and V_{ext} now being thread local and the *count* variable becoming an array indexing threads, i.e. $count[thread_{id}]$, in each leaf. Another relevant difference is that, when a new node in the g-trie needs to be created, its parent node has to be locked before creation. This is done to ensure that the same node is not created by multiple threads. Regarding work distribution, when a thread Q receives a work request from P, it needs to stop its computation, add the remaining work to W (line 8), split the work (line 9), give half of it to P (line 10) and resume its computation (line 11). After the enumeration phase is finished, the leafs are also distributed among the threads and isomorphism tests are performed to verify the appropriate canonical type of each occurrence in parallel (lines 14 to 17).

4.3 Work Request

When a thread P has completed its assigned work it asks a random thread Q for work. Random polling has been established as an efficient heuristic for dynamic load balancing [13] and, furthermore, in our case predicting exactly how much computation each active thread still has in its work tree can not be done without a serious overhead. If Q sends some unprocessed work, then P computes the work it was given. If Q did not have work to share, P tries asking another random thread. When all threads are trying to get work and no more work units are left to be computed, the enumeration phase ends.

4.4 Work Sharing

When a thread is computing and receives a work request, the execution is halted and work sharing is performed. In Figure 2 we show a work tree of a thread Q and its division with thread P. The work tree is built by the recursive calls to `addWork()`. The squares represent V_{used} and the current position in the g-trie. We only need the V_s of the deepest level since the parent g-trie nodes share the same vertices. The dotted nodes are work-units still to be explored. Note that these nodes are not stored in the g-trie, and they will be explored by the threads after sharing is performed and are presented only to give a more accurate view of the complete work tree generated by `FaSE`.

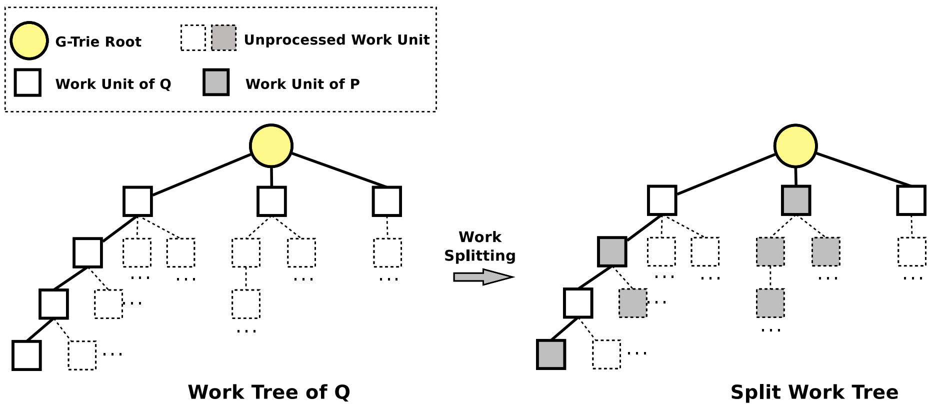

Fig. 2: The constructed work tree of a thread Q and its division when a work request is received from thread P.

During work division, each thread is given a g-trie level, constituted by V_s, V_{used} and the current g-trie position. In the given example, Q gets level 3 and 1 while P receives 2 and 4. The topmost level is fully split since that corresponds to the initial division from lines 4 to 6 of Algorithm 2.

4.5 Work Resuming

When work is shared the threads need a mechanism to resume their computation and that process is illustrates in Algorithm 3. The work levels are ordered from top to bottom (lines 2 and 3) so that only one V_s is necessary. If a work request is received, the general process of work sharing is performed (lines 4 to 8). No

Algorithm 3 Algorithm for resuming work after sharing is performed.

```
1: procedure RESUMEWORK($W$)
2:     ORDERBYLOWEST($W$)
3:     for all level $L$ of $W$ do
4:         if WORKREQUEST($P$) then
5:             $(W_Q, W_P) \leftarrow$ SPLITWORK(W)
6:             GIVEWORK($W_P$, $P$)
7:             RESUMEWORK($W_Q$)
8:             return
9:         if $L.depth = 0$ then
10:            for all vertex $v$ of $L.V_{ext}$ do
11:                ENUMERATE($\{v\}, \{u \in N(v) : u > v\}, T.root$)
12:        else
13:            ENUMERATE($L.V_s, L.V_{ext}, L.current$)
14:    ASKFORWORK()
```

call to `addWork()` is necessary since the work was either added previously to W before the current `resumeWork()` call or was added by the recursive `addWork()` calls from `enumerate()`. If the level being computed is the root of the g-trie, the top vertices are individually computed (lines 9 to 11), in the same manner as line 12 of Algorithm 2. Otherwise, the stored values of V_s, V_{used} and *current* are used to continue the previously halted computation (lines 12 and 13). If the thread finishes its alloted work it asks for more work (line 14).

5 Experimental Results

Experimental results were gathered on a 64-core machine; its architecture 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 consists of sets of two cores sharing a 2MB L2 and a 64KB L1 instruction cache. Every single core has a dedicated 16KB L1 data cache. The turbo boost functionality was disabled because that would lead to 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 over a dozen real-world networks and present here the results for a representative subset of them. In Table 1 a general view of the content and dimension of the chosen seven networks is shown. To showcase the general scalability of our algorithm, networks that vary in their field of application, their use of edge direction and their dimension were chosen. To decide what k to use, we simply opted for choosing one that gave a sufficiently large sequential time for parallelism to be meaningful but not so large that it would take more than a few hours to complete the computation.

Table 1: The set of seven different representative real networks used for our parallel performance testing.

Network	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$\frac{\lvert E(G)\rvert}{\lvert V(G)\rvert}$	*Directed*	*Description*	*Source*
`jazz`	198	2,742	13.85	No	Collaborations of jazz musicians	[1]
`polblogs`	1,491	19,022	12.76	Yes	Hyperlinks of politics weblogs	[2]
`netsc`	1,589	2,742	1.73	No	Network experiments co-authorship	[2]
`facebook`	4,039	88,234	21.85	No	Facebook friend circles	[3]
`company`	8,497	6,724	0.79	Yes	Media companies ownership	[4]
`astroph`	18,772	198,050	10.55	No	Astrophysics papers collaborations	[3]
`enron`	36,692	367,662	10.02	Yes	Email network	[3]

Table 2: General execution information and results.

Network	*Subgraph size*	*#Leafs found*	*#Subgraph types found*	*Sequential time (s)*	*#Threads: speedup*			
					8	*16*	*32*	*64*
`jazz`	6	3,113	112	295.95	**6.75**	**14.86**	**29.92**	**49.74**
`polblogs`	6	409,845	9,360	1,722.55	**7.85**	**15.56**	**30.04**	**47.48**
`netsc`	9	445,410	14,151	295.12	**7.83**	**15.05**	**23.82**	**26.54**
`facebook`	5	125	19	3,598.41	**7.67**	**15.34**	**31.00**	**51.81**
`company`	6	1,379	310	739.12	**7.94**	**15.81**	**31.02**	**48.53**
`astroph`	4	17	6	179.47	**6.62**	**13.60**	**24.69**	**30.42**
`enron`	4	17	6	1,370.46	**7.70**	**13.32**	**25.44**	**35.85**

To have the parallel version with one thread performing similarly to the sequential algorithm, work queues were not artificially created. This choice lead to a very small overhead (less than 5% for all tested cases) and, henceforth, parallel execution with one thread will be referred to as the *sequential time*.

Our algorithm's performance was evaluated up to 64 cores and results are presented in Table 2. In that table, the size of the subgraphs being queried, along with the number of g-trie leafs (the intermediate classes) and the actual number of different subgraph types are shown. The sequential time and the obtained speedups for 8, 16, 32 and 64 cores are also shown.

The results are promising and close to linear speedup up to 32 cores for every case. Due to the machine's architecture we did not achieve linear speedups for 64 cores but still managed to obtain a high efficiency for 4 of the 7 cases. Note that our algorithm performs worse in networks where many leafs need to be created. This problems arises because an unique g-trie is used and must be protected when a new node, leaf or label is inserted. Cases were found where speedups were severely limited by this factor. On the other hand, using one g-trie per thread would lead to much redundant work that would deteriorate our

[1] Arenas: `http://deim.urv.cat/~aarenas/data/welcome.htm`
[2] Mark Newman: `http://www-personal.umich.edu/~mejn/netdata/`
[3] SNAP: `http://snap.stanford.edu/data/`
[4] Pajek: `http://vlado.fmf.uni-lj.si/pub/networks/data/`

algorithm's performance. Memory also becomes a concern when many threads are used because each leaf has an array to store the frequencies. This limits the size of the subgraphs and networks that can be run. Another problem related to storing the frequencies in the g-trie is that it can sometimes lead to false sharing since many threads could be updating the array at the same time. A better option would be to instead have each thread keep an array of the frequencies for each leaf but, since the g-trie is created during runtime, the total number of leafs is not known and setting a unique *id* for each one would require resorting to locks. Finally, it was observed that memory allocations became heavier when more threads were used. Something that could be further explored is an efficient pre-alocation of memory, where the threads would retrieve it when needed. Furthermore, an adjacency matrix was used to represent the input network that, while giving the best possible algorithmic complexity for verifying node connections, imposes a quadratic memory representation. Different memory allocators, like `jemalloc` and `tcmalloc`, were tried but found no significant performance improvement.

By comparison, we have previous work parallelizing a set-centric approach with g-tries for multicore architectures [2] and obtained almost linear speedup for every case we tested. Besides using a conceptually different base algorithm (here we follow a network-centric algorithm). The main difference between the two approaches is that, in [2], the g-trie was pre-created before subgraph counting, removing the need to have locks when modifying the g-trie and making it possible to have subgraph frequencies outside of the g-trie, thus eliminating false sharing.

6 Conclusion

In this paper we presented a scalable parallel algorithm for the subgraph census problem. At the core or our method lies the `FaSE` algorithm, an efficient network-centric sequential approach which is able to drastically reduce the number of isomorphism tests needed when comparing to previous approaches such as `ESU` or `Kavosh`. `FaSE` induces a highly unbalanced search tree with independent branches and we devised a dynamic load balancing scheme capable of an efficient redistribution of work during execution time. We tested our algorithm on a set of representative networks and we achieved an almost linear speedup up to 32 cores and a high efficiency for the total 64 cores of our machine. To the best of our knowledge, this constitutes the fastest available method for a network-centric approach, allowing users to expand the limits of subgraph census applicability, not only on more dedicated computing resources, but also on their personal multicore machines.

In the near future it is our intention to explore a hybrid methodology capable of mixing both shared and distributed memory approaches. We also intend to carefully examine the possibility of using GPUs for computing a subgraph census. Finally, on a more practical angle, we will use our method to analyze several data sets, searching for new subgraph patterns that can lead to novel insight into the structure of these real-life networks.

Acknowledgments

David Aparicio 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/BPD/81695/2011).

References

1. Afrati, F.N., Fotakis, D., Ullman, J.D.: Enumerating subgraph instances using map-reduce. In: IEEE 29th International Conference on Data Engineering (ICDE). pp. 62–73. IEEE CS, Los Alamitos, CA, USA (2013)
2. Aparicio, D., Ribeiro, P., Silva, F.: Parallel subgraph counting for multicore architectures. In: IEEE International Symposium on Parallel and Distributed Processing with Applications. IEEE CS (August 2014)
3. Grochow, J., Kellis, M.: Network motif discovery using subgraph enumeration and symmetry-breaking. Research in Computational Molecular Biology pp. 92–106 (2007)
4. Kashani, Z., Ahrabian, H., Elahi, E., Nowzari-Dalini, A., Ansari, E., Asadi, S., Mohammadi, S., Schreiber, F., Masoudi-Nejad, A.: Kavosh: a new algorithm for finding network motifs. BMC bioinformatics 10(1), 318 (2009)
5. Khakabimamaghani, S., Sharafuddin, I., Dichter, N.t., Koch, I., Masoudi-Nejad, A.: Quatexelero: An accelerated exact network motif detection algorithm. PLoS ONE 8(7), e68073 (07 2013)
6. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network Motifs: Simple Building Blocks of Complex Networks. Science 298(5594) (2002)
7. Paredes, P., Ribeiro, P.: Towards a faster network-centric subgraph census. In: Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. pp. 264–271. ACM, NY, USA (2013)
8. Pržulj, N.: Biological network comparison using graphlet degree distribution. Bioinformatics 26(6), 853–854 (2010)
9. Ribeiro, P., Silva, F.: G-tries: a data structure for storing and finding subgraphs. Data Mining and Knowledge Discovery 28, 337–377 (March 2014)
10. Ribeiro, P., Silva, F., Kaiser, M.: Strategies for network motifs discovery. In: IEEE International Conference on e-Science. pp. 80–87. e-Science (2009)
11. Ribeiro, P., Silva, F., Lopes, L.: Efficient parallel subgraph counting using g-tries. In: IEEE International Conference on Cluster Computing (Cluster). pp. 1559–1566. IEEE CS (September 2010)
12. Ribeiro, P., Silva, F., Lopes, L.: Parallel discovery of network motifs. Journal of Parallel and Distributed Computing 72, 144–154 (2012)
13. Sanders, P.: Asynchronous random polling dynamic load balancing. In: Algorithms and Computation, pp. 37–48. Springer (1999)
14. Slota, G.M., Madduri, K.: Fast approximate subgraph counting and enumeration. In: 42nd International Conference on Parallel Processing. pp. 210–219 (2013)
15. Wang, T., Touchman, J.W., Zhang, W., Suh, E.B., Xue, G.: A parallel algorithm for extracting transcription regulatory network motifs. IEEE International Symposium on Bioinformatics and Bioengineering pp. 193–200 (2005)
16. Wernicke, S.: Efficient detection of network motifs. IEEE/ACM Transactions on Computational Biology and Bioinformatics pp. 347–359 (2006)
17. Zhao, Z., Wang, G., Butt, A.R., Khan, M., Kumar, V.A., Marathe, M.V.: Sahad: Subgraph analysis in massive networks using hadoop. Parallel and Distributed Processing Symposium, International 0, 390–401 (2012)