J. Parallel Distrib. Comput. 72 (2012) 144–154

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Parallel discovery of network motifs

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

ARTICLE INFO

Article history:
Received 14 October 2010
Received in revised form
4 July 2011
Accepted 29 August 2011
Available online 29 September 2011

Keywords:
Parallel algorithms
Complex networks
Graph mining
Network motifs

ABSTRACT

Many natural structures can be naturally represented by complex networks. Discovering network motifs, which are overrepresented patterns of inter-connections, is a computationally hard task related to *graph isomorphism*. Sequential methods are hindered by an exponential execution time growth when we increase the size of motifs and networks. In this article we study the opportunities for parallelism in existing methods and propose new parallel strategies that adapt and extend one of the most efficient serial methods known from the Fanmod tool. We propose both a master–worker strategy and one with distributed control, in which we employ a randomized receiver initiated methodology capable of providing dynamic load balancing during the whole computation process. Our strategies are capable of dealing both with exact and approximate network motif discovery. We implement and apply our algorithms to a set of representative networks and examine their scalability up to 128 processing cores. We obtain almost linear speedups, showcasing the efficiency of our proposed approach and are able to reach motif sizes that were not previously achievable using conventional serial algorithms.

1. Introduction

Many natural structures are intuitively represented by complex networks, which have received increased attention by the research community in recent years [2,24]. In order to mine interesting features from these networks, several different measurements can be applied [6]. In 2002, Milo et al. [23] noted that some subnetworks appeared with a much higher frequency in the studied networks than it would be expected in similar randomized networks, i.e., with the same degree sequence. These overrepresented topological patterns were named as *network motifs*.

Network motifs are important in the analysis of networks from several domains, particularly in the biological domain [3]. For example, it has been demonstrated that they can have functional significance in transcriptional regulatory networks [36] or protein–protein interaction networks [1]. They have been applied in other biological areas, like brain networks [37] or food webs [18], and they are also significant in networks from other domains, like electronic circuits [15] or software architecture [38]. We should note that the usage of network motifs does have some criticism [39,11,14], but in this article we do not try to position ourselves in that conceptual discussion. Instead, we focus our attention on the algorithmic aspect of finding network motifs, in order to obtain more efficient methods that can bring new insight into this topic.

Finding network motifs is a computationally hard task, since at its core, we are basically dealing with the graph isomorphism problem [8]. Current methods are almost all sequential in nature and revolve around finding the frequency of all subgraphs of a determined size (i.e., doing a *subgraph census*), both in the original network and in a random ensemble of similar networks [30]. The problem is that the execution time increases exponentially when we increase the motif or network size. Sampling has been introduced by Kashtan et al. [17] as a way to trade accuracy for time spent, but the process can still be very time consuming. Analytical methods to estimate the significance of motifs are now appearing [21,27]. These methods would be able to escape the need for the ensemble of random networks and their respective census, but there is still a long path to be accomplished in order for them to be general and practical enough to be used. In any case, we would still have to compute the subgraph census on the original network.

One way of improving network motif discovery is to resort to parallelism. Work in this area is still very scarce and it could have a significant impact in many application areas. It could lead not only to a speedup in the network motif calculation, but also to the discovery of larger motifs in bigger graphs that were previously unsuspected due to efficiency and time constraint reasons.

In this article we analyze the opportunities for parallelism in the existing sequential methods, leading to a parallel strategy able to have almost linear speedup in the whole network motif discovery process. We propose several parallel strategies for distributed memory systems that adapt and expand one the most efficient serial methods [43,30]. We study the applicability of both centralized control (with a master–worker strategy [13]) and completely distributed control (with a randomized receiver

* Corresponding author.
E-mail addresses: pribeiro@dcc.fc.up.pt (P. Ribeiro), fds@dcc.fc.up.pt (F. Silva), lblopes@dcc.fc.up.pt (L. Lopes).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.08.007

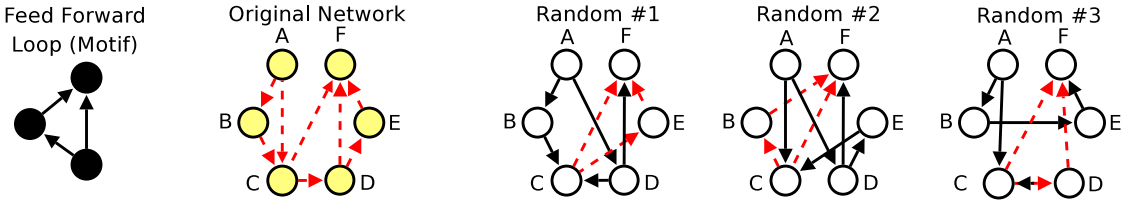

Fig. 1. An example motif of size 3. The three random networks present the exact same degree sequence as the original one, with each vertex preserving its in and out degrees. The feed forward loop, indicated by dashed edges, appears exactly three times in the original network ($\{A, B, C\}$, $\{C, F, D\}$, $\{D, E, F\}$), but only once in each random network. Note that different occurrences of the motif can share vertices and connections.

initiated strategy [33]). We use small independent work units that can be redistributed to provide dynamic load balancing both for exhaustive complete computations and approximations using sampling, efficiently parallelizing the whole network motif discovery problem. We implement our strategies using MPI and apply them on a series of representative networks. We obtain almost linear speedup up to 128 processing cores, demonstrating the efficiency and usability of our proposed method (from now on, the term core will be used to reference processing cores).

The reminder of this article is organized as follows. Section 2 establishes a network terminology, formalizes the problem we want to tackle and overviews related work. Section 3 points out the opportunities for parallelism and describes in detail our strategies for parallel network motifs discovery. Section 4 gives practical results when applying to a set of representative networks, studying the scalability. Section 5 concludes the article, commenting the results and describing possible future work.

2. Preliminaries

2.1. Network terminology

In order to have a well defined and coherent network terminology throughout the article, we first review the main concepts and introduce some notation that will be used in 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 vertices 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.

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. The *exclusive neighborhood* of a vertex v relative to a subgraph G is defined as $N_{exclusive}(v, G) = \{u \in N(v) : u \notin G \cup N(u)\}$.

A *mapping* of a graph is a bijection where each vertex 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.

2.2. Network motif problem

We start by formally defining the exact problem that we are trying to solve. For the sake of simplicity, from now on we will refer to network motifs simply as motifs. The motifs concept has several variations. We refer the reader to our previous work [30] or Ciriello and Guerra [5] for a more detailed view on this. In this article we will concentrate on the standard definition established by Milo et al. in the original article on motifs [23], and which is widely used in the literature.

Basically motifs are patterns of interconnections occurring in a complex network in significantly higher numbers than in similar random networks. Given a network, we want to calculate all of its motifs of size k as defined below:

Definition 1 (*k-Network Motif*)**.** An induced k-subgraph of a graph $G(k \leq |V(G)|)$ is considered a network motif if its frequency of occurrence in G is statistically overrepresented in relation to its frequency in R random graphs with the same degree sequence.

In order to determine the frequency of a specific subgraph, we also use the canon definition by Milo et al. [23], allowing arbitrary overlapping of vertices and connections between different occurrences of a subgraph. Fig. 1 illustrates this frequency concept. Schreiber and Schwobbermeyer [35] introduce different notions of frequency that could potentially be used when calculating motifs.

Typically the over-representation is calculated taking into account the subgraph frequency in the original network ($f_{original}$) and the mean subgraph frequency in the random networks ($\bar{f}_{random}$), by calculating the probability defined in Eq. (1).

$$\text{Prob}(\bar{f}_{random}(G_k) > f_{original}(G_k)) \leq P. \tag{1}$$

P is the probability threshold used for defining when a subgraph is considered a motif (0.01 is the value used by Milo et al. [23]). Again, there are several possible small variations (see [30,5]), including the need for a minimum frequency ($f_{original}$ must be bigger than a pre-defined parameter) and minimum deviation ($f_{original}$ must be sufficiently apart from $\bar{f}_{random}$), but what matters is that this measure can be analytically calculated in $O(R)$ time for a determined subgraph knowing its frequency both on the original and randomized networks.

If one can only have an approximate value of the frequencies (for example, obtained by sampling and not by exhaustive search), then we can use subgraph concentrations (see Eq. (2)) to derive an approximation of the required probability,

$$\text{Concentration}(G_k) = \frac{f(G_k)}{\sum_{\text{all subgraphs } i} f(G_k^i)}. \tag{2}$$

2.3. Related work

2.3.1. Sequential methods

We are aware of seven different main algorithmic strategies for finding motifs, all based on serial algorithms: `mfinder` [23], ESU [43], FPF [35], Grochow and Kellis (`grochow`) [9], `G-Tries` [28], `Kavosh` [16] and `MODA` [25].

Conceptually, these strategies are divided in three main categories:

- *network-centric methods*, that produce a *k*-subgraph census by enumerating all subgraphs of size *k* of the original graph (`mfinder`, FPF, ESU and `Kavosh`).
- *single-subgraph methods*, that excel in computing the frequency of a pre-defined single individual *k*-subgraph (`grochow` and `MODA`).
- *subgraph-set methods*, that specialize in counting the frequency of a set of pre-defined *k*-subgraphs, that do not necessarily have to be all of the possible *k*-subgraphs (`G-Tries`).

Regarding network-centric methods, `mfinder` was the original one, appearing in 2002, and it is based on a recursive backtracking algorithm that generates all *k*-subgraphs. Due to potential subgraph symmetries, the same subgraph can be found several times, since the search procedure can be started on any of its constituent nodes. `MAVisto`, from 2004, excels in using different frequency concepts and has an algorithmic performance comparable to `Mfinder` on the canon definition of frequency. ESU is an improved algorithm first introduced in 2005 that only allows searches being initiated on the nodes with an index higher than the root node. This breaks symmetries and each subgraph is found only once, leading to a more efficient method that can be sometimes orders of magnitude faster than `Mfinder`. `Kavosh` appeared in 2009, uses a novel counting algorithm that also exhibits a symmetry breaking behavior and appears to be asymptotically equivalent to ESU in algorithmic execution time terms.

In what concerns single-subgraph methods, `Grochow` introduced this concept in the motif finding problem in 2007 and uses a set of pre-built conditions to break symmetries. `MODA`, appearing in 2009, uses a pattern growth approach that starts with *k*-trees in order to arrive at a complete *k*-subgraph and also looks for non-induced subgraphs. It achieves considerable speedups when compared to `Grochow` for some types of *k*-subgraphs.

Finally, `G-Tries`, which constitute previous work in this field from our team and was first published in 2010, implements a subgraph-set methodology that relies on a specialized tree-like data structure able to identify common substructure on a set of *k*-subgraphs, using that to search at the same time the frequency of several subgraphs. `G-Tries` can achieve considerable speedups when compared to ESU and `Grochow`.

In a previous work from 2009 we provided pseudo-code and a practical empirical comparison of three of the algorithms described above (`mfinder`, ESU and `Grochow`) on a common platform, showing that ESU has the best general performance in exhaustive and complete subgraph census for different complex networks [30]. Note that at that time `Kavosh`, `MODA` and `G-Tries` did not exist.

All of the above mentioned methods, except `Grochow` and `G-Tries`, have publicly available source code (we expect to soon release the G-Tries source code). Three of those provide free graphical production software tools: `mfinder` [23], `Fanmod` [44] (which implements ESU) and `MAVisto` [43] (which implements FPF).

The described methods all have the option to produce exact exhaustive results. Some of them allow trading accuracy for speed, by using sampling. This concept was introduced for finding motifs in 2004 by Kashtan et al. [17], as an improvement to the `mfinder` tool. It relies on a procedure capable of extracting a single random *k*-subgraph, that is repeated a desired number of times. One problem with their approach is that the sampling is biased in the sense that not all subgraphs have the probability of being randomly chosen. `Fanmod` also has a sampling option that avoids that problem by extending the ESU algorithm in order to only show a fraction of the whole enumeration, obtaining unbiased sampling and guaranteeing that the same subgraph will never be sampled twice. This algorithm is called `ESU-Rand` and it is notably faster than the `Mfinder` sampling process [43]. However, it only allows pre-determination of an approximate number of samples wanted, in opposition to `Mfinder`, in which we can get the exact number of subgraphs we desire. In our previous work we give pseudo code for these two sampling algorithms [30]. `G-Tries` also has a unbiased sampling option similar to ESU [29]. The other algorithm allowing sampling is MODA, and relies on a probability distribution that is derived from the degree distribution of the original network. However, `ESU-Rand` still emerges as the fastest algorithm for the general case [25].

2.3.2. Parallel methods

Research work on parallel algorithms for motif discovery is still very scarce. Specific to the motif discovery problem and its associated subproblems, we are only aware of four distinct implemented and studied parallel approaches [31,41,32,34]. All four approaches focus essentially on parallelizing a single complete census. Parallelism on the whole motif discovery problem can then only be achieved by calculating consecutively the parallel census of the original and random networks, making it necessary to have synchronization after each census. The process of randomly generating a set of similar networks is also done sequentially. This clearly distinguishes these algorithms from ours, since we try to parallelize the complete motif discovery process and not just a single census. We also show how to parallelize the motif discovery by sampling, and not just by exhaustive enumeration.

We will now describe in more detail the existing approaches. Schreiber and Schwobbermeyer [35] do refer they were working in implementing a parallel version of their algorithm, but they defer the scalability analysis and further studies to future work that, to our best knowledge, has not yet been done or published.

Wang et al. [41] rely on finding neighborhood assignments for each vertex that avoids overlapping and redundancy on subgraph counts (as in the ESU algorithm), and try to balance the workload before the computation begins using mostly the node degrees. However, they do not detail the static scheduling process and they do not study the scalability of their approach, limiting the empirical analysis to a single network (the *E coli transcriptional* regulation network), and a fixed number of cores (32). Another characteristic of their approach is that they do not do isomorphism tests during the parallel computation, they wait until the end to check all the subgraphs and compute the corresponding isomorphic classes.

In a previous work [31], we also parallelize a single complete subgraph census. As in this work (see Section 3.2), we used the ESU algorithm as the starting point and extended it to obtain parallelization. However, two main differences separate that work from this one. First, we only parallelized a single subgraph census, and not the whole motif discovery process, which makes it necessary to synchronize the consecutive census and the sequential generation of random networks. Second, we used only a master–worker strategy [13] with a centralized work-sharing scheduling framework. In opposition, here we parallelize the whole motif discovery computation while also providing a completely distributed and dynamic `receiver-initiated` scheduling strategy, in a work-stealing framework.

In other previous work from us [32] we adapted and parallelized the `g-tries` subgraph matching algorithm, which computes the census of a pre-defined set of subgraphs. Again, it is different from what we are doing in this work, since we used `g-tries` as an underlying structure and parallelized only a single census.

Schatz et al. [34] focuses instead on parallelizing the Grochow and Kellis [9] approach. They do different single subgraph queries at the same time on different cores in a master–worker strategy, experimenting with static pre-computed scheduling (the same number of queries for each core) and first-fit scheduling (meaning that workers only process one query at a time and when they finish it they ask the master for more work). The latter has been

shown to have almost linear speedup against the corresponding sequential algorithm up to 64 cores, although it should be noticed that they only used one experimental network and in terms of motif discovery, following the Grochow and Kellis [9] approach, they would have to query the entire set of possible k-subgraphs, even when some of those subgraphs may not even appear on the networks. They have also tried to parallelize a single query using a network partition algorithm that creates small overlapping regions, adding some overhead for possibly repeated computations. This lead to some speedup, but again it was only tested on a single network and for eight different 7-subgraph queries.

2.3.3. Remark about other related methods

We should notice that motif discovery is conceptually quite different from *frequent subgraph discovery* [19,12], which derives from the *frequent itemset problem* [26]. In the former, we are looking for subgraphs that occur at least a pre-determined number of times in a minimum number of elements of a set of graphs. This is in opposition to really determining the frequency in a single original graph. Because of this, the algorithmic techniques used are different and parallelism on the frequent subgraphs problem is not directly applicable to our problem, even when sometimes the authors use the term motifs to designate the frequent subgraphs, like in the work of Wang and Parthasarathy [40].

3. Parallelizing motif discovery

We will now describe our strategies for parallelizing the whole network motif discovery. Section 3.1 describes the sequential workflow of the best known algorithms and points out the opportunities for parallelism. Section 3.2 introduces the adapted and extended serial algorithm that we will parallelize, showcasing work units as independent and atomic pieces of computation. Section 3.3 details our proposed parallel algorithm, describing the main workflow, the pre-processing phase, the main work phase, the final aggregation phase and the adaptation for parallelizing the approximation version of the algorithm, using subgraph sampling.

3.1. Serial workflow and opportunities for parallelism

The basic workflow of a serial motif discovery program, such as `Mfinder` or `Fanmod`, is depicted in the algorithm of Fig. 2. Basically, it revolves around doing a k-subgraph census on the original network (line 1) and then exhaustively creating a series of R similar random networks (line 3) and doing a k-subgraph census on each of them (line 4). After that, knowing the frequency of each existing isomorphic subgraph on all networks, its significance is calculated (line 5).

We identify the opportunities for parallelization that exist in the above algorithm, and define a taxonomy for later reference and use:

- *Census Parallelization*: do a complete census of all subgraphs of a determined size in parallel (lines 1 and 4). This could be done in several ways:
  - *Partition*: the network is pre-divided in several (possibly overlapping) partitions/regions and different cores analyze different partitions.
  - *Tree*: a recursive search procedure is executed in parallel, with different search tree branches being searched at the same time in different cores.
  - *Query*: on single-subgraph algorithms, each individual subgraph query is done separately by different cores.
- *Random Network Parallelization*: distribute the random networks between the cores, e.g. if we have to generate 100 random networks and have access to 100 cores, then each core could compute its own random network and its corresponding census.

Require: Graph G and integers k and R
Ensure: Motifs of size k in graph G
1: subgraphCensus(k, G)
2: **for** $i \leftarrow 1..R$ **do**
3: $R_i \leftarrow$ generateSimilarRandomNetwork(G)
4: subgraphCensus(k, R_i)
5: calculateSignificanceMotifs()

Fig. 2. Algorithm for serial network motif discovery.

Require: Graph G and integer k
Ensure: Complete k-subgraphs census of graph G
1: **for all** $v \in V(G)$ **do**
2: $V_{Ext} \leftarrow \{u \in N(v) : u > v\}$
3: EXTENDSUBGRAPH($\{v\}, V_{Ext}, v$)
4: **procedure** EXTENDSUBGRAPH(V_{Subg}, V_{Ext}, v)
5: **if** $|V_{Subg}| = k$ **then**
6: FOUND(V_{Subg})
7: **else**
8: **while** $V_{Ext} \neq \emptyset$ **do**
9: remove random chosen $w \in V_{Ext}$
10: $V'_{new} \leftarrow \{u \in N_{excl}(w, V_{subg}) : u > v\}$
11: $V'_{ext} \leftarrow V_{ext} \cup V'_{new}$
12: EXTENDSUBGRAPH($V_{subg} \cup \{w\}, V'_{ext}, v$)

Fig. 3. ESU algorithm.

- *Significance Parallelization*: distribute the significance calculation after the census is computed.

With this nomenclature we can now classify the strategies described in Section 2.3. All of them only use census parallelization. In order to do that, Wang et al. [41] rely on partition parallelization, we [31,32] rely on tree parallelization and Schatz et al. [34] discuss separately partition and query parallelization.

If we profile the serial algorithm during real computations, we find that the main bottleneck is the census computation, taking on average more than 95% of the whole execution time. Therefore, census parallelization is really a key issue and if we can do a single census in parallel, one way of doing the whole computation is to do exactly as the serial algorithm, except that the census calls are done in parallel. On its own, and as referred in Section 2.3, this strategy presents two main drawbacks:

- Synchronization is necessary after each census, ensuring that all cores have completed before the next census begins. Since typically we generate at least dozens of random networks, this can provoke a significant amount of unwanted idle time on cores.
- The other steps of the network motif discovery must be done sequentially. In particular, repeating the process of generating a similar network for every new random network can be time consuming.

We could therefore do better if we parallelized at the same time all the steps needed to discover motifs (not just the census) and this is precisely what we do in our proposed strategies. More than that, we also provide a version capable of using sampling in order to trade some accuracy for an improved execution time behavior.

3.2. Work units and a new serial algorithm

As a starting point we will use one of the most efficient methods known that also allows for sampling [30], which is the ESU algorithm. Fig. 3 details how ESU works.

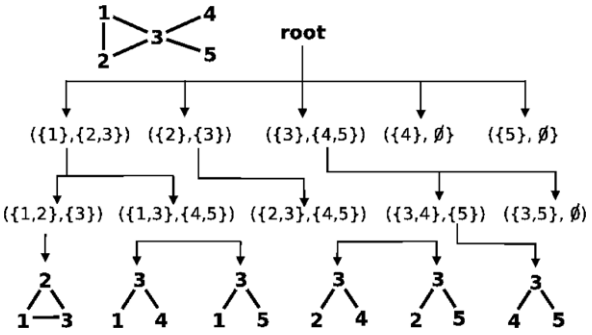

Fig. 4. The ESU algorithm generates this search tree for the given undirected network. Each tree internal node indicates the vertex sets passed as parameters to the procedure `extendSubgraph`, respectively V_{Subg} and V_{Ext}.
Source: Taken from Ribeiro et al. [30].

Its core idea is that we only expand a partially constructed subgraph with nodes that have a greater index number than the initial root node (lines 2 and 10). A list of possible extension vertices is therefore built (lines 2 and 11) and whenever a vertex is chosen to be extended, it is removed from the list (line 9). Its exclusive neighbors are then added to the new possible extensions (line 10). The fact that they are exclusive guarantees that each subgraph is enumerated exactly only once, because the ones which are not exclusive will be added on another instance of the recursion. Fig. 4 exemplifies in detail how the algorithm enumerates all 3-subgraphs of a graph with 5 vertices, showing how the algorithm transposes to a search tree.

Important to notice is that calls to `extendSubgraph()` are independent from each other and each one generates a set of other `extendSubgraph()` calls until we reach the desired subgraph size. Therefore, we define a single call to `extendSubgraph()` as the smallest piece of work we will use. This call is an ESU work unit (EWU) that is completely defined by a tuple of four values, as defined in Eq. (3).

$$EWU = (G_{Id}, V_{current}, V_{expandable}, v_{root}). \tag{3}$$

G_{id} is a network identifier (represented by a number: 0 (zero) for the original network, i for the i-th random network) and the three other tuple values correspond exactly to the `extendSubgraph()` parameters, and therefore map to a node in the tree of Fig. 4. From now on we will also be representing an EWU by a square $\square_{(id,s)}$ where id and s represent respectively the network identifier and the size of the partially constructed subgraph ($|V_{current}|$).

Besides this, we will represent the calculation of the census in a single network by a circle $\bigcirc_{id}$ where id represents a network identifier and the calculation of the significance of a single subgraph g by a diamond $\diamond_{sg}$. Note that different $\bigcirc_{id}$ are independent from each other, and so are $\diamond_{sg}$.

Require: Work Units Queue Q and motif size k
Ensure: A new updated Q and motif dictionary Dic

```
1:  procedure processWorkUnit(W)
2:      if type(W) = ◯_id then
3:          for i ← 1 ... |V(G)| do
4:              I_extension ← {u ∈ N(i) : u > i}
5:              □_(id,i) ← (id, {i}, I_extension, i)
6:              Q.pushFront(□_(id,i))
7:      else if type(W) = □_(id,s) then
8:          (id, V_subg, V_ext, v) ← □_(id,s)
9:          if s == k then
10:             count[id][canonicalString(V_subg)]++
11:         else
12:             while V_Ext ≠ ∅ do
13:                 remove random chosen w ∈ V_Ext
14:                 V'_new ← {u ∈ N_excl(w, V_subg) : u > v}
15:                 V'_ext ← V_ext ∪ V'_new
16:                 □_(id,s+1) ← (id, V_subg∪w, V'_ext, v)
17:                 Q.pushFront(□_(id,s+1));
18:     else if type(W) = ◇_g then
19:         if isSignificant(G_g) then
20:             report(G_g)
```

Fig. 6. Processing a single work unit.

Solving the network motif problem can, from now on, be expressed using our work units symbols. If we have R random networks with N different classes of isomorphic induced subgraphs, what we need is to solve in order the work queues Q_{census} and $Q_{significance}$, defined in Eqs. (4) and (5).

$$Q_{census} = \{\bigcirc_0, \bigcirc_1, \bigcirc_2, \ldots, \bigcirc_R\} \tag{4}$$

$$Q_{significance} = \{\diamond_0, \diamond_1, \diamond_2, \ldots, \diamond_N\}. \tag{5}$$

Our processing of the work units takes into account that each $\bigcirc_{id}$ can be decomposed in several smaller $\square_{(id,i)}$. So, for example, G_0 can be decomposed in $\{\square_{(0,1)}, \square_{(0,2)}, \ldots, \square_{(0,|V(G)|)}\}$. Every $\square_{(0,1)}$ will then be decomposable in several $\square_{(0,2)}$ and so on until the motif size is reached. Note that different $\square_{(id,i)}$ will generate a different number of smaller work units and have a completely unbalanced total execution time weight, corresponding to different topological parts of the network. Fig. 5 exemplifies the ensemble of work units that compose a network motif discovery problem.

Our algorithm for processing a single working unit is depicted in Fig. 6. If the work unit is a complete network ($\bigcirc$), it does generate a $\square$ for each of the network nodes (lines 3–6). This is the equivalent to executing the main procedure of the ESU algorithm as depicted

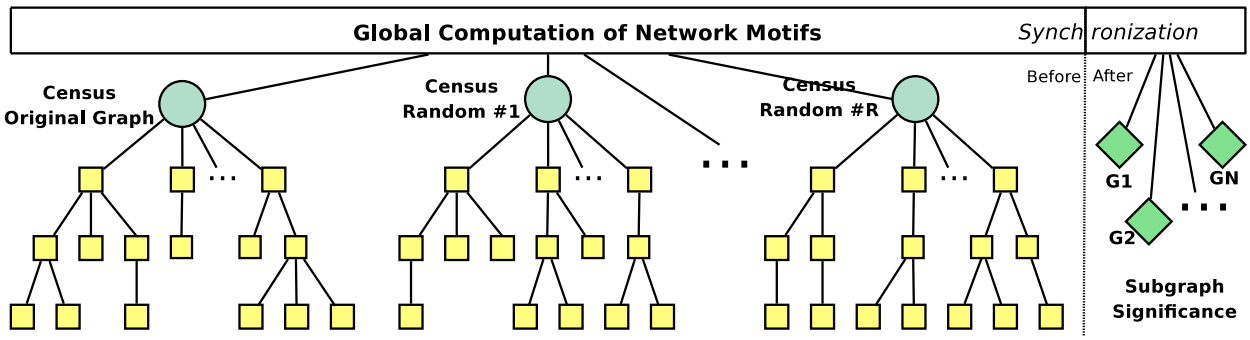

Fig. 5. An example motif computation of 3-motifs. The circles represent a complete census of a graph, the square an ESU Work Unit (a partially constructed subgraph) and the diamond the statistical significance of a k-subgraph. The census trees are completely unbalanced and that different random networks can lead to completely different sized trees. The significance can only be calculated after all the census are completed.

```
Require: Graph G, integers k and R
Ensure: Report k-motifs in G using R random networks
1: Q.clear()
2: for i ← 0...R do
3:     Q.pushBack(◯_i)
4: while Q.notEmpty() do
5:     processWorkUnit(Q.popFront())
6: for all G_i : do
7:     if count[0][G_i] > 0 then
8:         processWorkUnit(◇_Gi)
```

Fig. 7. New serial algorithm using defined work units.

in Fig. 3. If the work unit is a partially constructed subgraph (□), then it processes it exactly as in the ESU algorithm (8–17), except that instead of recursively calling itself, it puts all the expanded subgraphs in the work queue. Finally, if it is a significance work unit (◇), it calculates its significance and reports the subgraph when it is considered a motif (lines 18–20).

One more aspect needs additional explanation: line 10 of the algorithm. We we use a canonical string representation to deal with isomorphic subgraphs. We opted to use McKay's *nauty* algorithm [22], a widely known fast and practical implementation of isomorphism detection, to generate a canonical labeling and then use its adjacency matrix. In order to store the frequencies, we maintain a dictionary structure `count[id][sg]` where `id` stands for the graph identifier and `sg` for the subgraph. This dictionary could be implemented in many ways, and we currently use C++ STL's `map`, which is implemented using a balanced red–black tree.

With all of this, discovering network motifs in serial can now be done using the following algorithm (Fig. 7):

Lines 2 and 3 construct Q_{census}, lines 4 and 5 process it and lines 6–8 create and process $Q_{significance}$ (cf. Eqs. (4) and (5)). Note that we pop work units from the front of the queue, and since we push new work units also to the front, we traverse the search space in depth-first order. Therefore, analysis of random network #1 only starts after the original, random network #2 only after #1 and so on.

3.3. Parallel algorithms

Now that we have defined simple work units, the EWUs, we are ready to show our parallel strategies. In addition to the already mentioned variable, we will use P to represent the number of cores available.

The challenge is to divide the EWUs among all worker cores and our goal is to keep all workers busy, always computing some unprocessed EWU. Since the ESU tree is really unbalanced, a static division of the EWUs is not adequate to obtain a balanced load among all workers. It is preferable to have a dynamic load balancing scheme that constantly adapts during runtime and keeps redistributing work among the workers.

3.3.1. Main workflow and parallel strategy

There are three main steps in our algorithms, done in this order:

1. *Pre-Processing Phase*: in this phase we do all the necessary calculations in order to start our work, providing an initial work queue for each CPU.
2. *Work Phase*: in this phase we really do the bulk of the work, analyzing subgraphs and discovering their frequency. We will propose two different algorithms for this phase.
3. *Aggregation Phase*: in this phase we aggregate the subgraph frequencies found in each CPU and calculate the motif significances.

Our proposed strategies use basically the same pre-processing and aggregation phases and differ on the work phase, with two different possibilities:

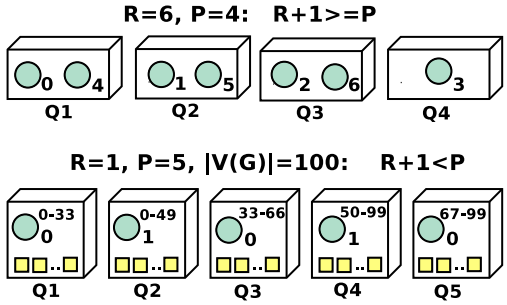

Fig. 8. Example of two pre-processing static divisions of work units among cores. When there are more networks than cores we divide the networks among the cores. When the opposite happens, we do the reverse and divide the cores among the networks, and then divide the nodes of networks equally between the cores. In both cases we use a round-robin scheme.

1. *Master–Worker strategy*: in this case there is a worker dedicated exclusively for the load balancing and work distribution (the master) and all the other workers process work and communicate only with the master in what regards the work phase. From now on we will call this strategy `master-worker`.
2. *Distributed strategy*: in this case all workers are responsible both for the work itself and the load balancing. At any time they can communicate with any other worker and dynamically try to redistribute the work. We will call this strategy `distributed`.

More details on this will be given in the next sections.

3.3.2. Pre-processing phase

Each core has its own work queue. In the case of `distributed` this means all cores, and in the case of `master-worker` this means all the cores except the master. Our first step is to initialize each of these work queues (let Q_i be the work queue of core i, $1 \leq i \leq P$). We need to make sure that all computational work units are in some work queue, i.e., $\cup Q_i = Q_{census}$. We identified two basic alternatives for this task:

- *all-in-one*: $Q_1 = Q_{census}$ and $\vee i \neq 1$, $Q_i = \emptyset$. This initializes the entire work queue on a single core and when the computation starts, all other cores must steal work from it.
- *static division*: We divide Q_{census} among all cores. We preferred this option since then all cores can start working on their work queues, without the need for initial communication between cores. In order to do that we do the following:
  - If $R + 1 \geq P$ we allocate networks (◯) to the cores in round-robin fashion.
  - If $R + 1 < P$, we allocate the cores in round-robin fashion to the networks (◯). We will end up with at least some networks having more than one core allocated, and we divide the network *id* initial nodes ($\square_{(id,1)}$) equally between the respective cores ($\bigcirc_{id}^{a-b}$ means we generate only ESU work units with root nodes from a to b).

Fig. 8 gives a practical example of how the static division scheme would work in practice. No matter what option we take for the initial division of work, whenever a core empties its work queue, it will immediately try to obtain more work, as you will be able to see in Section 3.3.3. What is important here is to give some initial more or less balanced work to all cores, in order to avoid unnecessary communication in the beginning of the computation.

3.3.3. Work phase

After having established an initial work queue we are ready to start doing the bulk of the computation.

Master–Worker strategy

In this case, there is a worker dedicated exclusively to perform load balancing and all other workers do the main work phase.

```
1:  procedure MASTER
2:      while not all workers are idle do
3:          msg ← ReceiveMessage(AnyWorker)
4:          if msg.type = RequestForWork then
5:              if Q.notEmpty() then
6:                  EWU = Q.popFront()
7:                  sendMessage(msg.Sender, EWU)
8:              else
9:                  IdleWorkers.pushBack(msg.Sender)
10:         else if msg.type = NewWorkUnit
    then
11:             if IdleWorkers.notEmpty() then
12:                 worker =IdleWorker.popFront()
13:                 sendMessage(worker,
    msg.EWU)
14:             else
15:                 Q.pushBack(msg.EWU)
16:     BroadcastMessage(Terminate);
```

Fig. 9. Work Phase procedure for master core within the master–worker strategy. Q is the list of unprocessed EWU's and *idleWorkers* the list of workers that are currently waiting for work.

```
1:  procedure WORKER
2:      while notFinished do
3:          if Q.empty then
4:              sendMessage(master,
    RequestForWork)
5:              msg ← ReceiveMessage(master)
6:              if msg.type = Terminate then ex-
    itWhile
7:              Q.pushBack(msg.EWU);
8:          PROCESSWORKUNIT(Q.popFront())
9:          if checkSplitThreshold() then
10:             while Q.hasMoreThanOneElement
    do
11:                 sendMessage(master,
    Q.popBack())
```

Fig. 10. Work Phase procedure for worker within master–worker strategy. Q stores the local work queue.

This is in its essence an extension to what we did previously in [31], but then it was only applied to a single subgraph census. The algorithms of Figs. 9 and 10 detail our strategy.

Basically the master keeps receiving messages from the workers (line 3) and acts accordingly. If the message is a request for more work (line 4) then it can either send the first unprocessed EWU (lines 6 and 7) or, if its work queue Q is empty, add the worker which sent the message to the list of workers that are in need of work (*idleWorkers*). If the message contains a new unprocessed EWU, then it either adds it to the work queue for future processing (line 15) or sends it directly to an idle worker, in case there is one (lines 11–13). If all workers are idle, it means that there are no more unprocessed EWUs and therefore we can end the work phase (line 2) and broadcast a termination message to all workers (line 16).

Regarding the worker execution, while there is no termination message (lines 2 and 6) it keeps processing its own work queue Q (line 8). If this queue is empty, then it asks the master for new work (line 4) and waits until a message is received (line 5), adding the newly received EWU to the local queue. Whenever `splitThreshold` is reached (line 9), the worker gives all but one of its unprocessed work units to the master, in order for them to be distributed among workers that are or will become idle. Note that this threshold is very important. If it is set too high, the work units will not be sufficiently divided in order to adequately balance

```
1:  while notFinished() do
2:      if Q.isEmpty() then
3:          askForWork()
4:      PROCESSWORKUNIT(Q.popFront())
5:      if checkRequestsThreshold() then
6:          serveWorkRequests()
```

Fig. 11. Work Phase main procedure.

the work among all workers. If it is too low, work will be divided too soon and the communication costs will increase. We tried two different options for the threshold: either a time limit or a number of EWUs processed limit. Section 4 gives more details on this.

Distributed strategy

In this case the load balancing decisions are distributed among all cores and every one of them, a worker, runs the algorithm of Fig. 11.

The basic idea is simple: while a worker still has work in its work queue, it keeps processing work units (line 4). If its queue becomes empty, then it asks for work from another worker (lines 2 and 3) and continues processing the new work units. If at a point in time a consensus is reached that the computation is over, the worker stops (line 1). The other key component is serving work requests from other workers (lines 5 and 6).

We will now explain in more detail the work request mechanism. The first thing to notice is that due to the nature of our desired environment (distributed memory with message passing) there is no way to steal work from another core without intervention from it. We must send a message and wait for an answer. All cores have a polling mechanism and from time to time (line 5, `checkRequestsThreshold()` function) they will check if there are any incoming requests. This threshold is important and can impact performance. If it is set too low, the receiver worker will be checking for messages too often and will spend valuable execution time trying to serve nonexistent requests. If it is set too high, the sender worker will have to wait for new work while remaining idle, because the receiver will take some time to check for messages.

We tried two different options for the threshold value: either a time limit or a number of EWUs processed, as in the `splitThreshold` of the master–worker. Section 4 details our experiments, that lead us to opt for a threshold based on the number of units processed.

We will now explain from which worker should we try to steal work from. Ideally we would know the worker that still has more work to do, but since in a completely distributed environment, that is not possible without incurring a major computation overhead. Besides, since the ESU trees are unbalanced, we cannot even have a precise prediction of our own work queue. Therefore we opted to always choose to ask a random worker for work, which was established as an adequate heuristic [33].

The third aspect to detail is our strategy for dynamically sharing work in a distributed setting. The main question here is to decide exactly which work units from our work queue should we share whenever a work request is received. The ideal option is to divide as equally as possible the work, in order to maximize the time in which both workers will not need to ask for work again. In order to do that we opted for a diagonal work-queue splitting scheme. Basically we distribute work units in the following way: one for the sender, one for the receiver, one for sender, and so on. As we are exploiting the search tree in a depth-first order, this will distribute as evenly as possible the work units, taking into account that work units at the same search depth will have similar computational costs. Of course that since the tree is really unbalanced, this cannot promise equal execution time, but it constitutes our best prediction implemented by a simple

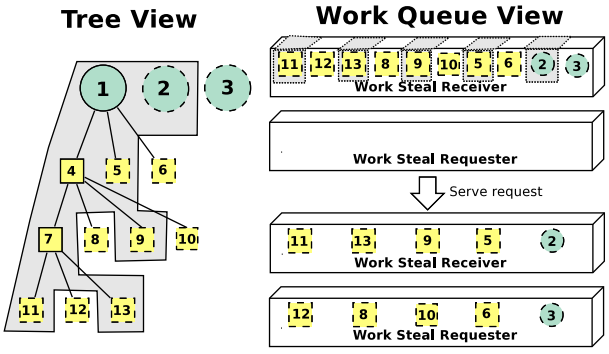

Fig. 12. Example of a diagonal work-queue splitting scheme. Dashed work units are yet to be explored. The shaded area corresponds to the work units that will remain in the receiver of the request. The other nodes go to the requester.

yet elegant solution, diagonally distributing along the search tree. Fig. 12 exemplifies our splitting scheme.

We will now explain our termination mechanism. Whenever someone asks for work and receives as response that the receiver's work queue is empty, it will ask another random worker, making sure that it does not repeat workers until new work units are found. If it happens that everyone answers that it has no more work, the worker will be able to conclude that indeed there is no more work and it will broadcast a "termination" message to everyone, ending this phase of the network motif discovery computation.

3.3.4. Aggregation phase

After the work-phase has ended, every worker will have its own dictionary of frequencies for every network analyzed (the original one and the random ones) and subgraphs discovered. There will probably exist many zeros, meaning that a particular network was not analyzed at all by that particular worker, but potentially there can be valuable information for every worker, for any subgraph, on any network. This is a huge a amount of data that we need to aggregate in order to calculate the subgraph significance. Note that the number of possible k-subgraphs grows super-exponentially as k increases.

For each class of isomorphic subgraphs, we need to know the frequency in the original network and its average frequency and standard deviation in the set of random networks. We choose to have a "root" worker, responsible for storing the global results. After gathering all needed frequencies, this worker can calculate the necessary subgraph significance in serial (note that, as said, the majority of the work is done during the computation of the census and that at this phase, since a computation of a single subgraph significance is done in constant time, it would not improve if it was sent to another worker—on the contrary it would take more time).

A simple primitive approach for this would be for each worker to communicate in turn with the root worker, sending its own results. In order to do that, it would have to send pairs of subgraph descriptions (for example using the canonical labeling) and their respective frequencies. We call this strategy `naive`.

This naive approach is not enough and we improved it. Our strategy is to first agree on the list of subgraphs that are being computed. If all workers have that list before communicating its own frequencies found, we could avoid the need to communicate graph identifications, because we can induce a fixed order of relevant subgraph types. Therefore, if we communicate a vector of frequencies, the position in the vector will determine which subgraph we are referring too.

In order to create that list of relevant subgraph types, the root worker will start by advertising a set of T concrete subgraphs that it knows to occur in the original subgraph, by broadcasting a message (note that a root worker will always have in its work queue at least part of the original network). Then, all workers, organized in a binary tree, communicate to tree ancestors their list of subgraph types not found by the root worker. After this process is complete, the root will have a list of new subgraphs that it will broadcast to all workers.

Require: Work Units Queue Q and motif size k
Ensure: A new updated Q and
1: approximate motif dictionary Dic
2: **procedure** PROCESSWORKUNIT(W)
3: **if** TYPE(W) $= \bigcirc_{id}$ **then**
4: lines 3 to 5 of the algorithm in Figure 6
5: With probability P_i: Q.pushFront($\square_{(id,i)}$)
6: **else if** TYPE(W) $= \square_{(id,s)}$ **then**
7: lines 8 to 17 of the algorithm in Figure 6
8: With probability P_{s+1}: Q.pushFront($\square_{(id,s+1)}$)
9: **else if** TYPE(W) $= \diamond_g$ **then**
10: lines 19 and 20 of the algorithm in Figure 6

Fig. 13. Processing a single work unit with sampling.

Having the pre-defined list at hand, the workers can just communicate frequencies (and not graph descriptions). Instead of using point-to-point messages (from one worker to another worker), we use the specialized MPI collective communications facilities. We use `MPI_Reduce` to gather and sum all frequency values in a vector. The position in the vector denotes which network and subgraph type it refers to. We therefore delegate to the MPI implementation the gathering of information, by explicitly denoting that we want to aggregate and sum the frequencies.

We call this strategy of first reducing the information that we need to communicate (by agreeing on the list of subgraphs and communicating it using an underlying binary tree) `hierarchical` and then use MPI primitives for collective aggregation.

3.3.5. Parallel sampling

The ESU algorithm also allows sampling and we have implemented that option, providing a quicker but only approximate parallel calculation of motifs. The basic idea is to only follow each search branch with a probability related to the depth. Fig. 13 details our sampling version of `processWorkUnit`, emulating `RAND-ESU`, the sampling version of the `ESU` algorithm.

The algorithm is basically the same with the exception of lines 5 and 8, that establish that each EWU is only explored with a certain depth-related probability. When looking for $k-$ *subgraphs*, each possible occurrence will only be found with probability $P_0 \times P_1 \times \cdots \times P_{(k-1)}$.

This is directly parallelizable using the described algorithms and essentially adds an even more unpredictable search tree topology, since a branch can sometimes be completely eliminated because it was not selected in the respective probabilistic test.

4. 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 3.8 TB of disk space, using Infiniband interconnect. For our experiments, we had access to a maximum of 128 cores. The code was developed in C++ and compiled with gcc 4.1.2. For message passing we used OpenMPI 1.2.7. All the times measured were *wall clock times* meaning real time from the start to the end of all processes.

Table 1
Networks used for experimental testing of our parallel strategy.

Network	Nodes	Edges	Avg. degree	Directed	Description
`foodweb` [10]	37	203	5.48	Yes	Food web middle Chesapeake Bay in summer
`social` [20]	62	159	2.56	No	Social network of a dolphins community
`neural` [42]	297	2345	7.89	Yes	Neural network of *C. elegans*
`metabol` [7]	1057	2527	2.39	Yes	Metabolic network of *S. pneumoniae*
`protein` [4]	2361	7182	3.04	No	Protein–protein interaction network of *S. cerevisiae*
`power` [42]	4942	6549	1.32	No	USA western states power grid

Table 2
Maximum achievable subgraph sizes k with a serial program for computing motifs using 100 random graphs.

Network	k	Execution time (s)	Average growth	Approx. total nr subgraphs	Nr isomorphic classes
`foodweb`	8	24 363	9.2 ± 2.7	2×10^9	478 654
`social`	8	7 428	8.1 ± 2.0	1×10^9	4 940
`neural`	5	23 082	46.5 ± 0.8	1×10^{10}	7 072
`metabol`	5	113 676	81.5 ± 3.3	4×10^8	24
`protein`	5	30 572	31.9 ± 1.8	7×10^9	21
`power`	7	10 842	6.12 ± 1.4	7×10^8	626

In order to evaluate our parallel algorithms, we used six different representative networks, from different domains and with different topological features. These are summarized in Table 1.

The number of possible parameter choices is too vast to show all its correspondent results in this article. Therefore we had to make some choices. The first one is related to the random graphs. For our first batch of tests, we opted to fix the number R of random graphs to 100, a quantity capable of already giving meaningful results in terms of subgraph significance (and used in many articles, such as [37]). The random graphs were generated by exchanging each edge 3 times similarly to what is done in Fanmod.

We want to show that our strategies are able to effectively parallelize the problem at hand, so we need to use cases where the sequential execution time is big enough to justify parallelization up to the 128 cores. For that purpose we measured the sequential execution time for computing all motifs of size k as we increase k, and we stopped when that time was larger than one hour. We also calculated the average growth ratio, that is, for increasing values of k, how much the execution time grew. We also calculated the approximate total number of different occurrences of k-subgraphs in all the graphs and the number of different subgraph types (isomorphic classes) found in the original network. The results obtained, using algorithm 3, can be seen in Table 2.

The average growth ratios have a relatively low standard deviation (the number after $\pm$), which means we can make rough estimates on how much more time we would need to compute increased greater motif sizes. Note also that growth differs a lot from network to network, and is not directly related to the size of the network. Typically, directed networks present a larger growth, since they implicitly represent more connectivity options and more possible subgraph types.

Before showing results from the whole parallel motif computation, we will first show results for the aggregation phase, in order to establish the best option for this part of our strategy. For the fixed networks used and for the k subgraph size chosen (Table 2), we show the time it takes solely for aggregating in the root the frequencies of all subgraph types that appear in the original network. We show the two alternatives described in Section 3.3.4: the `naive` and the `hierarchical` approach.

Table 3 details the results we obtained, using from 8 up to 128 cores (with one core, no aggregation is needed).

As expected, it takes more time to aggregate results when there are more isomorphism classes, i.e., there are more different types of subgraphs to communicate. Generally speaking, the naive approach always seems to be worse. When we double the number of cores we roughly need twice the amount of time.

Table 3
Aggregation times (in seconds) for two different approaches with 100 random networks and 8–128 cores.

Network	k	Method	#CPUs: time spent (s)				
			8	16	32	64	128
`foodweb`	8	`naive`	6.23	47.31	129.44	294.43	624.51
		`hierar.`	2.98	3.65	4.66	4.92	9.77
`social`	8	`naive`	0.05	0.43	1.28	2.98	6.37
		`hierar.`	0.02	0.16	0.34	0.36	1.33
`neural`	5	`naive`	0.06	0.37	1.09	2.51	5.39
		`hierar.`	0.04	0.08	0.47	0.47	0.97
`metabol`	5	`naive`	0.01	0.03	0.04	0.07	0.14
		`hierar.`	0.00	0.02	0.04	0.03	0.03
`protein`	5	`naive`	0.02	0.03	0.04	0.07	0.15
		`hierar.`	0.00	0.04	0.04	0.03	0.04
`power`	7	`naive`	0.03	0.06	0.14	0.34	0.71
		`hierar.`	0.01	0.11	0.06	0.07	0.08

This can limit the scalability and it is a serious problem if we have more frequencies to communicate (for example, by using more random networks or by searching larger motifs). Using our `hierarchical` strategy, we can overcome this linearity and cut the time logarithmically. Note especially the case of the `foodweb` network, with a really significant improvement from the `naive` execution time. Based on these results we can see that our collective approach performs better in the general case, and from now on we will assume all results are obtained using it.

We can now show the global behavior of our parallel algorithms, described in Section 3.3, both with master–worker and distributed strategies. One important aspect of both algorithms are the thresholds. Initially we thought of using time spent as the respective unit, but we discovered that precious time was being wasted by calling the operating system time functions. We then empirically verified that using the number of work units processed would give better results, and we proceeded by obtaining threshold values that performed generally well for our computing environment. In our case, we ended up using units equivalent to 0.5 s of processing time for both our strategies: in the case of the master–worker for the splitting threshold; in the case of distributed for the check requests threshold. What we want to show is that our parallel algorithms are viable, efficient and scalable options for motif discovery. Table 4 details the speedups we obtained, i.e., the relative time gain when compared to the sequential version.

Both algorithms present very good performances and scale well up to 128 cores. The distributed strategy is the better option since all cores can be used for the computation, with no CPU power wasted. Note that the algorithms are very flexible and work well

Table 4
Parallel performance of the algorithm. The speedup is measured when compared to the sequential version of algorithm 3.

Network	k	Strategy	#CPUs: speedup				
			8	16	32	64	128
foodweb	8	Master–Worker	6.8	14.6	29.6	59.8	117.4
		Distributed	7.8	15.4	30.9	61.7	118.1
social	8	Master–Worker	6.6	14.1	29.2	59.0	117.5
		Distributed	7.7	15.4	31.1	61.3	120.2
neural	5	Master–Worker	6.6	14.7	29.9	59.9	123.3
		Distributed	7.7	15.3	31.3	60.9	124.4
metabol	5	Master–Worker	6.6	14.6	30.1	60.7	120.9
		Distributed	7.7	15.5	31.2	61.4	123.2
protein	5	Master–Worker	6.7	14.3	29.7	60.4	121.3
		Distributed	7.8	15.6	31.3	61.9	122.4
power	7	Master–Worker	6.6	14.5	29.2	61.4	119.0
		Distributed	7.6	15.4	30.7	61.4	122.9

Table 5
Parallel performance with 128 cores as the number of random networks change. The speedup is measured when compared to the sequential version of algorithm 3.

Network	k	Nr of random networks			
		0	10	100	1000
foodweb	8	124.05	121.57	118.1	106.71
social	8	122.98	121.37	120.1	114.45
neural	5	123.24	123.96	124.4	121.36
metabol	5	124.32	124.29	123.2	122.46
protein	5	122.91	122.62	122.4	122.47
power	7	123.06	124.62	122.9	124.05

Table 6
Parallel performance of the sampling version of the algorithm. In all networks we sampled 10% of the k-subgraphs by using probabilities $(1, 1, \ldots, 1, 0.1)$.

Network	k	#CPUs: speedup				
		8	16	32	64	128
foodweb	8	7.8	15.1	30.3	59.5	105.9
social	8	7.6	15.3	30.7	61.6	115.7
neural	5	7.7	15.4	31.3	61.8	120.6
metabol	5	7.7	15.7	31.0	62.0	123.8
protein	5	7.7	15.6	30.6	61.6	123.6
power	7	7.8	15.3	31.2	62.1	121.8

both for cases where the number of random networks is greater than the number of CPUs, and when this number is smaller. In order to further verify this claim, we used the full 128 cores on all networks using the distributed strategy, while varying the number of random networks. The most extreme lower case is when we have zero random networks (in that case we are merely doing a subgraph census of the original network, with no statistical significance) and we extend the number of random networks up to 1000. Table 5 details the results we obtained.

The algorithm scales well for all cases, being able to create a good load balance, regardless of the variation in the number of networks. The speedup decreases as we increase the number of random networks since we are increasing the number of frequencies found and therefore we need to communicate and aggregate more data in the end.

As described in Section 3.3.5, the sequential ESU algorithm allows one to trade accuracy for execution time. We have proposed a parallel version of the sampling version of ESU (Fig. 13) and we experimented to sample 10% of all subgraphs, by using as probability parameters $(1, 1, \ldots, 1, 0.1)$, a customary setting nicknamed a "fine" version in [43], with a good trade-off between speed and accuracy. Table 6 details the results obtained (we used the distributed strategy).

As before, the algorithm adapts well to the even more unpredictable search tree shape and is able to maintain scalability. The speedup is relatively smaller than the one obtained with the complete exhaustive enumeration since with sampling we take less time to enumerate but we still need roughly the same time for the aggregation phase.

Given all the results shown, we can say that our parallel algorithm is very flexible and achieves near optimal scalability for all experimented settings of the network motif discovery problem. Our best option in terms of performance is to adopt a distributed strategy for load balance, an hierarchical strategy for the aggregation in the end, and a threshold value to attend work requests that depends on the number of work units processed. With these settings, we are able to obtain an almost perfect speedup on all our tested use cases.

A direct comparison with previous approaches is not feasible. However, by observing the results shown by Wang et al. [41] and Schatz et al. [34] we achieve better performance, study the scalability more comprehensively, and we use a larger number of cores.

Our results allow larger motif sizes that were not previously reachable. By observing again the average growth of execution time in Table 2, one can see that in all cases, using 128 cores will at least make it possible to compute motifs with one more vertex in the same amount of time. And in some cases even more. We tested it with the power network, and we were able to find all motifs of size 9 in approximately 1 h, something which was estimated to take almost a week using the basic sequential algorithm. Note that enabling larger motif sizes could provide new valuable information for the analysis of the respective network, even when the increase is just by one over the previously known motifs.

5. Conclusions

In this paper we studied the opportunities for exploiting parallelism when discovering network motifs. We provided novel parallel algorithms that adapt and extend one of the most efficient serial algorithms, the ESU algorithm. Our best parallel strategy is capable of dynamically dividing the work during the computation, using random polling, and of guaranteeing a really distributed load balance where all cores are working in the motif discovery process. We also provided a scalable strategy for aggregating the results in the end of the computation.

We tested our algorithms on a set of different representative networks and we achieved almost linear speedup up to 128 cores in all networks, for a vast amount of different settings. Our algorithm is even capable of parallelizing a sample version of the original sequential algorithm, which paves the way for even further performance gains.

We have shown that our methodology cannot only greatly speed up the discovery of motif of sizes already feasible, but can also allow for increasing both network and motif sizes to limits that were before unfeasible for practical reasons, due to the huge amount of time that was needed. This paves the way for the discovery of new potentially unknown motifs than can assume great importance. The applicability of our method is also very broad, and is not restricted to biological networks, but basically to whatever structure that can be represented as a complex network.

In the future, we plan to automatically and dynamically compute all threshold parameters, allowing the algorithm better adaptability for very different computing environments. We also want to give a more applicational view to our methodology, by effectively searching for bigger motifs on real networks that may be of interest. Finally, very soon we expect to release a working tool for discovering motifs based on our algorithms.

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). Finally, we would like to thank the reviewers for the valuable comments and suggestions.

References

[1] I. Albert, R. Albert, Conserved network motifs allow protein–protein interaction prediction, Bioinformatics 20 (18) (2004) 3346–3352.
[2] R. Albert, A.L. Barabasi, Statistical mechanics of complex networks, Reviews of Modern Physics 74 (1) (2002).
[3] E. Alm, A.P. Arkin, Biological networks, Current Opinion in Structural Biology 13 (2) (2003) 193–202.
[4] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, G. Li, R. Chen, Topological structure analysis of the protein–protein interaction network in budding yeast, Nucleic Acids Research 31 (9) (2003) 2443–2450.
[5] G. Ciriello, C. Guerra, A review on models and algorithms for motif discovery in protein–protein interaction networks, Briefings in Functional Genomics 7 (2) (2008) 147–156.
[6] L.F. Costa, F.A. Rodrigues, G. Travieso, P.R. Villas Boas, Characterization of complex networks: a survey of measurements, Advances in Physics 56 (2007) 167.
[7] J. Duch, A. Arenas, Community detection in complex networks using extremal optimization, Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) 72 (2005) 027104.
[8] P. Foggia, C. Sansone, M. Vento, A performance comparison of five algorithms for graph isomorphism, in: Graph Based Representations in Pattern Recognition, 2001.
[9] J. Grochow, M. Kellis, Network motif discovery using subgraph enumeration and symmetry-breaking, Research in Computational Molecular Biology (2007) 92–106.
[10] J. Hagy, Eutrophication, hypoxia and trophic transfer efficiency in Chesapeake Bay, Ph.D. Thesis, University of Maryland Center for Environmental Science, Horn Point MD, USA, 2002.
[11] J. Hallinan, P. Jackway, Network motifs, feedback loops and the dynamics of genetic regulatory networks, in: Proceedings of the IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, 2005.
[12] J. Han, M. Kamber, Data Mining: Concepts and Techniques, second ed., Morgan Kaufmann, ISBN: 1558604898, 2006.
[13] E. Heymann, M.A. Senar, E. Luque, M. Livny, Evaluation of an adaptive scheduling strategy for master–worker applications on clusters of workstations, in: Proceedings of the 7th International Conference on High Performance Computing, HiPC 2000, Bangalore, India, 2000.
[14] P.J. Ingram, M.P. Stumpf, J. Stark, Network motifs: structure does not determine function, BMC Genomics 7 (2006) 108.
[15] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, U. Alon, Coarse-graining and self-dissimilarity of complex networks, Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) 71 (Pt. 2–1) (2005).
[16] Z. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, A. Masoudi-Nejad, Kavosh: a new algorithm for finding network motifs, BMC Bioinformatics 10 (1) (2009) 318.
[17] N. Kashtan, S. Itzkovitz, R. Milo, U. Alon, Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs, Bioinformatics 20 (11) (2004) 1746–1758.
[18] M. Kondoh, Building trophic modules into a persistent food web, Proceedings of the National Academy of Sciences 105 (43) (2008) 16631–16635.
[19] M. Kuramochi, G. Karypis, Frequent subgraph discovery, in: IEEE International Conference on Data Mining, 2001, p. 313.
[20] D. Lusseau, K. Schneider, O.J. Boisseau, P. Haase, E. Slooten, S.M. Dawson, The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. can geographic isolation explain this unique trait? Behavioral Ecology and Sociobiology 54 (4) (2003) 396–405.
[21] C. Matias, S. Schbath, E. Birmelé, J.J. Daudin, S. Robin, Network motifs: mean and variance for the count, REVSTAT 4 (2006) 31–35.
[22] B. McKay, Practical graph isomorphism, Congressus Numerantium 30 (1981) 45–87.
[23] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, U. Alon, Network motifs: simple building blocks of complex networks, Science 298 (5594) (2002) 824–827.
[24] M.E.J. Newman, The structure and function of complex networks, SIAM Review 45 (2003).
[25] S. Omidi, F. Schreiber, A. Masoudi-Nejad, Moda: an efficient algorithm for network motif discovery in biological networks, Genes & Genetic Systems 84 (5) (2009) 385–395.
[26] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: ICDT'99: Proceedings of the 7th International Conference on Database Theory, Springer-Verlag, London, UK, ISBN: 3-540-65452-6, 1999, pp. 398–416.
[27] F. Picard, J.J.J. Daudin, M. Koskas, S. Schbath, S. Robin, Assessing the exceptionality of network motifs, Journal of Computational Biology (2008).
[28] P. Ribeiro, F. Silva, G-tries: an efficient data structure for discovering network motifs, in: ACM Symposium on Applied Computing, 2010.
[29] P. Ribeiro, F. Silva, Efficient subgraph frequency estimation with g-tries, in: International Workshop on Algorithms in Bioinformatics, WABI, in: LNBI, Springer, 2010.
[30] P. Ribeiro, F. Silva, M. Kaiser, Strategies for network motifs discovery, in: Proceedings of the 5th IEEE International Conference on E-Science, IEEE CS Press, Oxford, UK, 2009.
[31] P. Ribeiro, F. Silva, L. Lopes, Parallel calculation of subgraph census in biological networks, in: Proceedings of the 1st International Conference on Bioinformatics, Valencia, Spain, 2010.
[32] P. Ribeiro, F. Silva, L. Lopes, Efficient parallel subgraph counting using g-tries, in: IEEE International Conference on Cluster Computing, Cluster, IEEE CS Press, 2010.
[33] P. Sanders, Asynchronous random polling dynamic load balancing, in: International Symposium on Algorithms and Computation, 1999.
[34] M. Schatz, E. Cooper-Balis, A. Bazinet, Parallel network motif finding, 2008.
[35] F. Schreiber, 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.
[36] S.S. Shen-Orr, R. Milo, S. Mangan, U. Alon, Network motifs in the transcriptional regulation network of escherichia coli, Nature Genetics 31 (1) (2002) 64–68.
[37] O. Sporns, R. Kotter, Motifs in brain networks, PLoS Biology 2 (2004).
[38] S. Valverde, R.V. Solé, Network motifs in computational graphs: a case study in software architecture, Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) 72 (2) (2005).
[39] A. Vazquez, R. Dobrin, D. Sergi, J.P. Eckmann, Z.N. Oltvai, A.L. Barabasi, The topological relationship between the large-scale attributes and local interaction patterns of complex networks, Proceedings of the National Academy of Sciences 101 (2004) 17945.
[40] C. Wang, S. Parthasarathy, Parallel algorithms for mining frequent structural motifs in scientific data, in: ACM International Conference on Supercomputing, ICS, 2004.
[41] T. Wang, J.W. Touchman, W. Zhang, E.B. Suh, G. Xue, A parallel algorithm for extracting transcription regulatory network motifs, in: Bioinformatics and Bioengineering, IEEE International Symposium on, vol. 0, 2005, pp. 193–200.
[42] D.J. Watts, S.H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.
[43] S. Wernicke, Efficient detection of network motifs, IEEE/ACM Transactions on Computational Biology and Bioinformatics 3 (4) (2006) 347–359.
[44] S. Wernicke, F. Rasche, Fanmod: a tool for fast network motif detection, Bioinformatics 22 (9) (2006) 1152–1153.

Pedro Ribeiro is an Invited Auxiliary Professor (50% FTE) in the Computer Science Department of the School of Sciences at the University of Porto (UP), Portugal. He was recently awarded his Ph.D. in Computer Science from the University of Porto (2011). His primary research interests are in algorithms and data structures, parallel and distributed computing and complex network analysis. He also has a strong interest in Computer Science Education and Programming Contests.

Fernando Silva is an Associate Professor in the Computer Science Department of the School of Sciences at the University of Porto (UP), Portugal. He was awarded his Ph.D. in Computer Science from the University of Manchester, UK (1993), and, in 2007, obtained the Habilitation in Informatics from the New University of Lisbon, Portugal. He currently coordinates the Center for Research in Advanced Computing Systems (CRACS), and is a member of the scientific board of MAP-i, the Doctoral Program in Computer Science of the Universities of Minho, Aveiro and Porto (was director in 2008/09 edition). His primary research interests are in logic programming, programming languages, parallel and distributed computing, peer-to-peer, applications in information mining and bioinformatics. He has advised 8 completed Ph.D. theses in these areas.

Luís Lopes is an Associate Professor in the Computer Science Department of the School of Sciences at the University of Porto (UP), Portugal. He was awarded his Ph.D. in Computer Science from the University of Porto (1999) and his primary research interests are in programming languages (mainly domain specific), virtual machines, wireless sensor networks, distributed computing, and middleware. His latest project involves the development of type-safe, semantically robust, resource-aware, programming languages for wireless sensor networks.