FLIP-CPM: A Parallel Community Detection Method

Enrico Gregori[1], Luciano Lenzini[2], Simone Mainardi[1,2], and Chiara Orsini[1,2]

[1] Institute of Informatics and Telematics,
Italian National Research Council, Pisa, Italy
[2] Department of Information Engineering,
University of Pisa, Pisa, Italy
enrico.gregori@iit.cnr.it, l.lenzini@iet.unipi.it,
simone.mainardi@iet.unipi.it, chiara.orsini@iet.unipi.it

Abstract. Uncovering the underlying community structure of networks modelling real-world complex systems is essential way to gain insight both into their structure and their functional organization. Of all the definitions of community proposed by researchers, we focused on the k-clique community definition as we believe it best catches the characteristics of many real networks. Currently, extracting k-clique communities using the methods available in the literature requires a formidable amount of computational load and memory resources. In this paper we propose a new parallel method that has proved its capability in extracting k-clique communities efficiently and effectively from some real-world complex networks for which these communities had never been detected before. This innovative method is much less resource intensive than Clique Percolation Method and experimental results show it is always at least an order of magnitude faster. In addition, tests run on parallel architectures show a noticeable speedup factor, in some cases linear with the number of cores.

1 Introduction

The automatic discovery of network communities provides an insight into the mesoscale structure of real-world complex networks, which are far too large to be made sense of manually, even with the help of visualization techniques. There have been numerous definitions of community in the literature including k-clique communities [6], which are fine-grained and tightly connected. Unfortunately, their extraction requires a substantial amount of memory and computational load for large-scale complex networks. The first technique for extracting k-clique communities, called Clique Percolation Method (CPM), was proposed in 2005 by Palla *et al.* [6]. CPM can be broadly partitioned into the following three phases. The first consists in extracting all the maximal cliques of the input graph. Since the number of maximal cliques is exponential with the number of nodes in the graph, it is unlikely to find an algorithm with provably good execution times with reference to the number of nodes. An exhaustive review of these algorithms can be found for example in [1, Sect. 5]. The second phase of CPM consists in

building up a clique-clique overlap matrix, which was proposed by Everett in 1998 [3] as a tool for describing and analysing the amount of overlap between cliques. In the third phase CPM extracts k-clique communities by carrying out a component analysis of binary matrices obtained from the clique-clique overlap matrix. Unfortunately CPM does not scale in terms of space required and computational time. The first step toward the enhancement of CPM was made in 2008 by Kumpula *et al.* [5] with the Sequential Clique Percolation method (SCP). SCP was designed for discovering communities with a given k. However, as also highlighted by the authors, since it generates and processes cliques rather than maximal cliques, it is extremely slow on graphs with medium-large cliques (e.g. with more than 10 nodes). To the best of our knowledge, no software tool has so far been able to extract k-clique communities from the global Internet topology at the Autonomous System[1] (AS) level of abstraction, which is the main topic of our research activities. This encouraged us to design and develop an innovative parallel method capable of alleviating CPM drawbacks (i.e. space requirements and execution time). In Sect. 2 we illustrate this method, which processes the clique-clique overlap matrix in chunks of configurable size and in parallel analyses the overlap by exploiting a multi-processor computing architecture, which nowadays is available almost everywhere (e.g. laptops and standard desktops). We present experimental results in Sect. 3 and discuss conclusions and future work in Sect. 4.

2 Fast Lightweight Parallel CPM

To improve CPM performance, we developed an optimized parallel version of the method, named Fast LIghtweight Parallel Clique Percolation Method (FLIP-CPM), which relies upon: a) a new technique, called incremental G_k connected components; and b) a *sliding window* mechanism which enables parallelization. While a) is designed to strongly reduce memory requirements associated with the size of the clique-clique overlap matrix, b) is designed to reduce the execution time. At first, FLIP-CPM extracts and sorts all the l maximal cliques from the input graph G. Then, for each k between 3 and k_{max}, it creates an initial set containing as many singletons as the number of maximal cliques with size greater than or equal to k — k_{max} is the maximal clique maximum size in G. At this point FLIP-CPM slides the window, starting from the beginning down to the end of the clique-clique matrix and for each new window, it performs parallel operations described in paragraph b) hereafter.

a) Incremental G_k Connected Components. Assuming that, for each k, the binary matrix is an adjacency matrix of a graph G_k, k-clique communities are equivalent to the connected components of G_k, which in turn correspond to the k-clique connected components of G. The graph G_k is made up of $\sum_{i=k}^{k_{max}} l_i = n_k$ nodes, where l_k is the number of maximal cliques with size k in G. However, our

[1] A connected group of one or more IP prefixes run by one or more network operators that has a single and clearly defined routing policy.

goal is not to have a complete topology of G_k. In fact, although such a topology can tell a lot more about the interactions between cliques, in order to extract k-clique communities, the connected components of G_k are sufficient. Maintaining these (runtime growing) connected components can be easily classified as *partially* dynamic problems on undirected graphs [2, Chapt. 8]. This problem may be efficiently addressed with the use of the so called *set union* data structures [2, Chapt. 5]. These data structures enable us to maintain a collection of disjoint sets under an intermixed sequence of *union* and *find* operations, starting from a collection of singleton sets. *Union(h,i)* combines the two sets h and i into a new set. *Find(j)* returns the set containing element j. A detailed analysis of set union algorithms together with their worst-case execution time can be found in [7]. The technique we have designed, called *incremental G_k connected components*, is outlined here. For each of the $\binom{n_k}{2}$ possible combinations of maximal cliques, overlap is computed. Whenever at least $(k-1)$ nodes are found to be shared between a pair of maximal cliques, the corresponding nodes in G_k, let them be u and v, are checked via two find operations: $U \leftarrow$ *Find(u)* and $V \leftarrow$ *Find(v)*. If $U{=}V$, nodes are already connected and hence the overlap for another pair can be immediately processed. If $U{\neq}V$, nodes are in two separate components that have to be merged with a *Union(U,V)* before analysing the next pair of maximal cliques. With the technique described above, space complexity has a linear dependence, i.e. $O(l)$, on the number of maximal cliques, rather than quadratic as in CPM where the clique-clique overlap matrix is used.

b)Sliding Window and Parallelization. The basic idea behind the new method we designed is to process the clique-clique overlap matrix (hereinafter referred to simply as matrix) in parallel, in *chunks* of configurable size, through a *sliding window* on the matrix itself. The sliding window enables multiple threads to process the matrix as if it were in memory in its entirely, while actually only a *chunk* physically resides in memory. Let W be the size, in bytes, of the window. If w bytes are used for each element in the matrix, then the maximum number of elements that can be placed in the window is $\eta = \lfloor W/w \rfloor$, constant and known a priori. Furthermore, if s is the index of the first row in the window, there is a way to also compute, a priori, the index e s.t. the maximum number of consecutive rows can fit in the window; i.e. the rows with indices i s.t. $i \in [s, e]$. In fact, the range $[s, e]$ varies according to the position of the sliding window on the matrix. Assuming that the window is slid across the upper triangular part, excluding the diagonal, of the $l \times l$ matrix; if the rows have indices between 0 and $(l-1)$, we can solve the following equation:

$$\sum_{j=s}^{j=x} (l - (j+1)) = \eta. \tag{1}$$

After some straightforward algebra, taking into account that $\sum_{i=1}^{i=r} i = r(r+1)2^{-1}$ for each positive integer r, we can rewrite the (1) as $-\frac{x^2}{2} + x\left(l - \frac{3}{2}\right) + (s-1)\left(\frac{s-2l+2}{2}\right) = \eta$, finding it has two solutions for x: $x_1 = \frac{2l-3}{2} - \frac{1}{2}\sqrt{\Delta}$ and

$x_2 = \frac{2l-3}{2} + \frac{1}{2}\sqrt{\Delta}$, where $\Delta = (2s - 2l + 1)^2 - 8\eta$. It can be verified that (1) always has at least one real solution (i.e. $\Delta \geq 0$) if the following constraints hold: a) $l \geq 2$ because the problem only makes sense only if the number of maximal cliques is greater than 1; b) $\eta \geq l - 1$ because the window must be sized to contain, at least, the largest row, i.e. the one with index 0; c) $0 \leq s \leq l - 1$ since the starting index must be one of the possible indices of the matrix. So, assuming that the first row in the window has index s, the index e can be computed by solving (1) and considering, in order, the following cases: i) if $x_1 = l - 2$ and $x_2 = l - 1$, $e = x_2 = l - 1$; ii) else if $x_1 \geq 0$, $e = \lfloor x_1 \rfloor$; iii) else, i.e. $x_1 < 0$, $e = l - 1$ because all the rest of the matrix can be placed in the window. That said, it is easy to design an API that implements the following functions: i) $Slide(s)$ that, given s as input, computes and returns e; and ii) $Read(i, j)$ and $Write(i, j, value)$ which provide read/write access to the elements with indices i, j s.t. $i \in [s, e]$ and $i < j \leq l-1$. Such elements can easily be located in memory by adding the offset $w \left(\sum_{h=s}^{h=i-1} [l - (h + 1)] + j - i - 1 \right)$ to the first address of the window.

Chunks are processed by two pools of threads. First, a pool $\mathrm{T_B} = \{\tau_0, \ldots, \tau_{b-1}\}$ of b threads computes overlap values and writes them through the API discussed in the previous paragraph. Each thread is assigned a subset of rows in $[s, e]$ to process. That is, for each thread $\tau_t \in \mathrm{T_B}$, the subset consists of all the rows i in $[s, e]$ s.t. $t = i \pmod b$. Since the subsets are disjoint, there is no need for any mechanism to ensure that write operations are carried out in mutual exclusion. Furthermore, the previous round-robin like row assignment, although very simple, ensures that most likely multiple threads perform an almost equal number of operations if maximal cliques are ordered by decreasing (increasing) size. When each thread in $\mathrm{T_B}$ has finished processing its own rows, another pool $\mathrm{T_C} = \{\tau_0, \ldots, \tau_{c-1}\}$ of $k_{max} - 2 = c$ threads reads every row with index in $[s, e]$ and keeps the connected components of c graphs $G_3, G_4, \ldots, G_{k_{max}}$ updated accordingly, using the incremental G_k connected components technique previously described. That is, each thread $\tau_t \in \mathrm{T_C}$ is responsible for the incremental G_{t+3} connected components. Once again, mutual exclusion mechanisms have been avoided.

3 Experimental Results

In this section we show the experimental results obtained by running a C implementation of FLIP-CPM and CFinder [6], a free closed-source implementation of CPM. They have been run both on a standard personal computer, the iMac[1] and on a 4 CPUs (24 cores) server machine, the Dell[2].

In Tab. 1 we summarize some characteristics of the input graphs. Specifically, n, m and l are the number of nodes, the number of edges and the number of maximal cliques respectively. $\mu = l^{-1} \sum_k k \cdot l_k$ is the average maximal clique

[1] (2x) Intel E7600 CPU @ 3.06GHz; (4x) 2GB RAM modules @ 1067MHz; Mac OS X 10.6.4 operating system, Darwin 10.4.0 kernel

[2] (4x) Intel E7540 CPU @ 2GHz; (16x) 4GB RAM modules @ 1067MHz; GNU/Linux operating system; Linux 2.6.35.22 kernel

Table 1. Graphs Properties

	n	m	l	μ	σ^2	ρ
Internet	35,390	152,233	2,747,484	22.64	20.54	0.976
AMS-IX	322	13,434	752,108	22.20	9.55	1
LINX	345	14,188	384,494	23.01	11.29	1
NL-IX	224	2,619	4,127	11.09	5.89	0.976
MSK-IX	293	4,225	1,593	12.36	9.42	0.994
SwissIX	116	1,110	669	8.62	2.46	0.993
MIX-IT	76	861	714	8.32	2.50	0.939
KleyReX	119	932	332	8.08	5.63	0.968

size and $\sigma^2 = l^{-1} \sum_k l_k (k - \mu)^2$ the variance. Finally, ρ is the density of the clique-clique overlap matrix, i.e. the the ratio between the number of elements with a value greater than zero and the total number of elements. Figure 1(a) shows the computation time experienced by running both FLIP-CPM and CPM on the iMac. Several small Internet Exchange Point (IXP)-induced subgraphs were used as inputs. In fact they are the only IXP-induced subgraphs on which CFinder has enough memory to complete its execution. It was not possible to run it on the other graphs as the square of their number of maximal cliques is much larger than the amount of memory available on the iMac. The new method was always at least one order of magnitude faster.

Figure 1(b) shows the computation time experienced by running FLIP-CPM on the Dell, with an exponentially growing number of threads. Two large graphs were used as inputs. The dashed line in the figures represents the ideal case where doubling the number of threads implies halving the execution time (i.e. a speedup linear with the number of threads). With a number of threads less than or equal to eight, the new method achieves the best performance: in this range, the execution time decreases exponentially with the number of threads. When the number of threads becomes greater than or equal to sixteen, the speedup becomes less significant. FLIP-CPM seems to work well also on graphs much larger than the ones used in the previous paragraphs when running time analyses and comparisons. For example, an analysis of the k-clique communities of the Internet at the AS level, recently published in [4], has been carried out using FLIP-CPM. Moreover, we have been able to execute it on the Enron email network and the Condense Matter collaboration network in a few minutes.

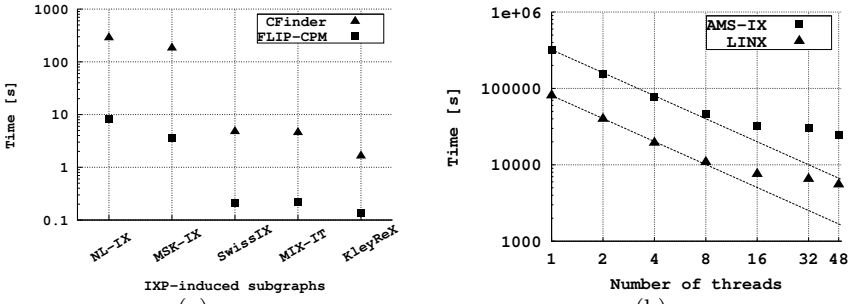

Fig. 1. Computation times experienced by running FLIP-CPM and CPM on our iMac using some small graphs as inputs (a) and computation time versus number of threads on AMS-IX and LINX, respectively (b).

4 Conclusions and Future Work

In this paper we tackled the problem of extracting k-clique communities from real-world complex networks. We have described Fast LIghtweight Parallel Clique Percolation Method (FLIP-CPM), which greatly reduces the memory required for the extraction of communities, through the use of special data structures that have a *linear* (rather than quadratic) dependence on the number of maximal cliques. The new method also reduces the execution time by exploiting a parallel computing architecture. With far fewer stringent requirements in terms of memory, the new method is shown to be experimentally able to extract, for the first time, k-clique communities also from the Internet AS-level topology graph. FLIP-CPM is highly effective and, even when run on standard hardware architectures, it turns out to be at least one order of magnitude faster than CPM. The effectiveness is confirmed by the speedup, in a multi-processor environment, which in several cases, proves to be linear with the number of cores. The efficiency can be boosted by parallelizing the algorithm which keeps the incremental connected components updated at runtime. We plan to carry out this parallelization as part of our future work. Furthermore the analysis of more and more dense and large complex systems, such as Wikipedia or Facebook networks, is very well under-way in order to test the capability of FLIP-CPM on networks of this scale.

References

1. Bomze, I.M., Budinich, M., Pardalos, P.M., Pelillo, M.: The maximum clique problem. In: Handbook of Combinatorial Optimization. pp. 1–74. Kluwer Academic Publishers (1999)
2. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Algorithms and Theory of Computation Handbook. CRC Press, Inc., 1st edn. (1998)
3. Everett, M.G., Borgatti, S.P.: Analyzing Clique Overlap. Connections 21(1), 49–61 (1998)
4. Gregori, E., Lenzini, L., Orsini, C.: k-clique Communities in the Internet AS-Level Topology Graph. In: SIMPLEX 2011 (2011)
5. Kumpula, J.M., Kivelä, M., Kaski, K., Saramäki, J.: Sequential algorithm for fast clique percolation. Phys. Rev. E 78(2), 026109 (Aug 2008)
6. Palla, G., Derenyi, I., Farkas, I., Vicsek, T.: Uncovering the overlapping community structure of complex networks in nature and society. Nature 435(7043), 814–818 (2005)
7. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM 31, 245–281 (March 1984)