FLIP-CPM: A Parallel Community Detection Method

E. Gregori, L. Lenzini, S. Mainardi, C. Orsini

IIT TR-31/2010

Technical report

Dicembre 2010

FLIP-CPM: A Parallel Community Detection Method

Enrico Gregori[1] Luciano Lenzini[2] Simone Mainardi[3]
Chiara Orsini[4]

December 22, 2010

[1]Institute of Informatics and Telematics, Italian National Research Council, Pisa, Italy. E-mail: `enrico.gregori@iit.cnr.it`

[2]Department of Information Engineering,University of Pisa, Pisa, Italy. E-mail: `l.lenzini@iet.unipi.it`

[3]Institute of Informatics and Telematics, Italian National Research Council, Pisa, Italy and Department of Information Engineering,University of Pisa, Pisa, Italy. E-mail: `simonemainardi@gmail.com`

[4]Institute of Informatics and Telematics, Italian National Research Council, Pisa, Italy and Department of Information Engineering,University of Pisa, Pisa, Italy. E-mail: `chiara.orsini@iet.unipi.it`

Abstract

Uncovering the underlying community structure of the Internet at the AS level is essential way to gain insight both into its structure and its 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 the Internet AS-level topology.

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 communities efficiently and effectively from real-world complex networks, including the Internet at the AS level. 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.

Chapter 1

Introduction

The automatic discovery of network communities provides an insight into the mesoscale structure of the Internet Autonomous System (AS) level topology graph [19], which is far too large to be made sense of manually, even with the help of visualization techniques (e.g. [1]). Revealing the organization of the Internet AS-level topology graph is important for many reasons [17], including the following:

- To interpret the global organization as the coexistence of its structural subunits associated with more highly interconnected parts, which may help in understanding the mechanisms responsible for its genesis and (possibly) its evolution;
- To uncover relationships between the ASs which are not apparent by inspecting the graph as a whole (e.g. ASs belonging to large densely-connected communities are very prone to interacting via peering relationships rather than via customer-provider relationships);
- To better understand dynamic processes (e.g. processes involved in a BGP AS path set up or spread of viruses), which may be dramatically affected by the modular structure of the graph;
- To classify ASs according to their structural position [11], for instance: ASs which have a central position within communities may have an important function of control and stability within the group, while ASs on the boundaries which belong to several communities may play an important mediation role.

Identifying and extracting these cohesive and a priori unknown building blocks (i.e. communities) is thus crucial for an understanding of the structural and functional characteristics of the Internet at the AS level or, more generally, of large-scale complex networks of which it is one example.

There have been numerous definitions of community in the literature including k-core [2], k-dense [23] and k-clique communities [20] together with the algorithms for their extraction. Specifically in [23] it was proved that a k-clique community implies a k-dense community, which in turn implies a k-core community.

The previous three community definitions seem to be suitable for exploring the structure of the Internet at least at the AS level of abstraction. In [23] it was shown that the method for extracting k-core communities, although computationally efficient [4], does not reveal a detailed community structure in the sense that it only discovers coarse-grained communities with very loosely interconnected groups. The k-dense method extracts communities almost as efficiently as the k-core method, while the qualities of extracted communities enable us to gain more insight into the Internet AS-level topology [14] graph than the k-core method. On the other hand, the k-clique method is the most accurate and can extract fine-grained and tightly connected communities. However it requires a substantial amount of memory and computational load for large-scale complex networks such as the Internet at the AS level.

The first technique for extracting k-clique communities from a generic graph, called Clique Percolation Method (CPM), was proposed in 2005 by Palla *et al.* [20]. CPM can be broadly partitioned into the following three phases. The first consists of reporting all the maximal cliques of the input graph. Since the number of maximal cliques is exponential with the number of nodes in the graph [18], 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 [5, Sect. 5]. Parallel algorithms were designed and proposed in [8] [24] [27]. The second phase of CPM consists of building up a clique-clique overlap matrix, which was proposed by Everett in 1998 [10] 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.* [16] 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 while running on graphs with slightly 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 AS-level topology. This encouraged us to design and develop an innovative parallel method capable of alleviating CPM drawbacks (i.e. space requirements and execution time). In this paper 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, shared memory, computing architecture, which nowadays is available almost everywhere (e.g. laptops and standard desktops).

The rest of this paper is organized as follows. In Chapter 2 we present a systematic analysis of CPM, highlighting its scalability. In Chapter 3 we propose a number of contributions that meet the former scalability issues and which highlight the innovative nature of our method. In Chapter 4, we show several experimental results, which demonstrate that the new method significantly outperforms CPM in terms of memory requirements and execution time. Finally, in Chapter 5, we discuss several related issues and future directions.

Chapter 2

Clique Percolation Method and Related Issues

Before describing CPM and its scalability issues, we need to introduce some definitions. A k-clique of a graph is a complete subgraph with exactly k nodes. A k-clique community is defined as the union of all the k-cliques that can be reached by each other through a series of adjacent k-cliques. Two k-cliques are adjacent if they have $(k-1)$ nodes in common. A more formal definition uses the concepts of the k-clique chain and k-clique connectedness. A k-clique chain is the union of a sequence of adjacent k-cliques. Two k-cliques are k-clique connected if they are part of the same k-clique chain. Therefore k-clique communities are equivalent to the k-clique connected components of a graph.
CPM can be logically and practically partitioned into three phases, namely: Maximal Cliques Listing, Clique-Clique Overlap Matrix Construction, and k-Clique Community Extraction. The complexity analysis and the aspects of these phases which are relevant to the paper are be discussed below.

Maximal Cliques Listing. CPM extracts maximal cliques using the algorithm described in [21, Sect. 1.1.2], which looks like a coarser version of the Bron-Kerbosh algorithm [6]. However, since there are at most $3^{\frac{n}{3}}$ maximal cliques in an n-nodes graph [18], it is unlikely that there are efficient algorithms for their enumeration. For example, any algorithm with an $O(3^{\frac{n}{3}})$ worst-case execution time, such as the one in [26], is the best one could hope for as a function of n. Fortunately, listing the maximal cliques in the current Internet AS-level topology graph was not an issue since it took a few minutes to completely extract even with a standard PC.

Clique-Clique Overlap Matrix Construction. Once the maximal cliques have been enumerated, CPM builds a clique-clique overlap matrix as in [10]. Each maximal clique is associated with a row (column) and the elements of the matrix represent the number of shared nodes between the corresponding maximal cliques. It follows that the clique-clique overlap matrix is symmetric and diagonal elements represent the size of the maximal cliques. Hence the upper (lower) triangular part, diagonal excluded, is enough to contain non-redundant information on the overlap.

Let l be the number of maximal cliques in the graph. It is clear that with a standard storage format, the space complexity of the matrix scales quadratically with l; this is in spite of simple optimizations that take into account, for example, the symmetry of the matrix. More efficient storage formats have been proposed for sparse matrices [3] but, unfortunately, experimental results have shown that the clique-clique overlap matrices are very dense (i.e. their elements are almost all non-zero), at least the one resulting from the Internet AS-level topology graph.

Assuming maximal cliques are memorized as ordered lists of integers representing the nodes, the worst-case execution time required for computing the matrix is $O(l^2 k_{max} \log_2 k_{max})$, where k_{max} is the size of a maximum clique. However in practice it is unlikely to find graphs where all the maximal cliques are also maximum and hence execution times can be significantly shorter than the worst-case scenario where all the cliques are k_{max} in size.

In this phase there are two scalability issues that make CPM inapplicable or, at least, extremely slow when running on graphs modelling real world complex systems such as the Internet at the AS level. Both are determined by a quadratic dependence on the number of maximal cliques, which can grow exponentially with the number of nodes in the graph. In the Internet AS-level topology graph resulting from our dataset, we were able to extract approximately $3 \cdot 10^6$ maximal cliques, with a size varying between 3 and 36. Thus, the first scalability issue regards the amount of space needed for storing the clique-clique overlap matrix which, as previously stated, is $O(l^2)$. With reference to the Internet AS-level topology graph, under the assumption that a single byte is enough for each element of the matrix, a rough estimation of the space required is $(3 \cdot 10^6)^2 = 9 \cdot 10^{12}$ B ≈ 8.2 TB.

The second scalability issue regards the amount of time required for processing each element (i.e. the number of nodes in common between any pair of maximal cliques) of the clique-clique overlap matrix. The most efficient way this can be done is with a binary search: although this is an efficient operation, it has to be done for each possible pair of maximal cliques. Hence, as regards the Internet AS-level topology graph, about $9 \cdot 10^{12}$ binary searches are needed. To solve the above two scalability issues, we designed a new efficient method which is reported in Chapter 3.

k-Clique Community Extraction. CPM extracts k-clique communities from binary matrices built from the clique-clique overlap matrix by: a) putting at 1 every on-diagonal element greater than or equal to k and every off-diagonal element greater than or equal to $(k-1)$; b) zeroing every other element, and then c) carrying out a component analysis of this matrix. However no technique for accomplishing a component analysis has been proposed in the CPM related literature and hence a worst-case complexity analysis of this phase would have little sense.

Chapter 3

Fast Lightweight Parallel CPM

To improve CPM performance, we developed an optimized 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; b) a parallel shared-memory architecture; and c) a *sliding window* mechanism. While a) and c) are designed to strongly reduce memory requirements associated with the size of the clique-clique overlap matrix, b) is designed to reduce the execution time.

3.0.1 Incremental G_k Connected Components

As discussed in Chapter 2, CPM extracts communities by carrying out a component analysis of binary matrices obtained from the clique-clique overlap matrix. 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.

Nevertheless, the graph G_k can be built by incrementally adding edges to it, rather than with the clique-clique overlap matrix. It can be created and runtime kept updated by sequentially processing the overlap between pairs of maximal cliques. Unfortunately, the overlap information, which was previously contained in the clique-clique overlap matrix, is now embodied in G_k. Since dense l-nodes graphs are $O(l^2)$ in terms of space, no improvement was achieved by the incremental construction of G_k.

Luckily, our 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 [9, Chapt. 8]. This problem may be efficiently addressed with the use of the so called *set union* data structures [9, 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 d singleton sets $\{0\}, \{1\}, \ldots, \{d-1\}$. Initially, the label of set $\{j\}$ is j. Moreover, the label of each set always corresponds to one of the items contained in the set itself. *Union(h,i)* combines the two sets h and i into a new set labelled h. *Find(j)* returns the label of the set containing element j. A detailed analysis of set union algorithms together with their worst-case execution time can be found in [25].

The technique we have designed, called *incremental G_k connected components*, is outlined below. At first n_k sets are created, one for each maximal clique that belongs to a (not yet fully discovered) k-clique community. Subsequently, 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 the labels match, i.e. $U = V$, nodes are already connected and hence the overlap for another pair can be immediately processed. If the labels do not match, i.e. $U \neq V$, nodes are in two separate components that have to be merged with a *Union(U, V)* before analysing the next pair of maximal cliques. When the overlap has been analysed for each pair, k-clique communities can be extracted via a find operation on each label of the initial n_k singleton sets. Let α be a very slowly growing function, a functional inverse of Ackermann's function. The former union and find operations can be performed in $O(\alpha(q, n))$ amortized time, where q and n are the total number of operations and the number of nodes in the graph respectively [9, Th. 8.3].

However, the most important result achieved with set union data structures together with the technique described above regards the space complexity. We were able to make its dependence linear, i.e. $O(l)$, on the number of maximal cliques, rather than quadratic as in CPM where the clique-clique overlap matrix is used.

3.0.2 Parallelization

The number of operations required to analyse the overlap between pairs of maximal cliques scales quadratically with their total number. Hence the previous analysis can be very time-consuming: in fact around 10^{12} pairs must be processed on the Internet AS-level topology graph. This issue can be addressed if the overlap is computed in parallel by a pool of processes. In the following discussion the terms process, flow of execution and thread are used interchangeably.

Although in 3.0.1 we showed a technique that made the clique-clique overlap matrix unnecessary, we were working under the implicit assumption of a single flow of execution. If, by exploiting a parallel shared-memory computing architecture, multiple flows of execution are available, this matrix is useful again to enable cooperation between multiple threads. The basic idea behind the new method we designed is to process the matrix in parallel, in *chunks* of configurable size, through a *sliding window* on the matrix itself.

Sliding Window. The sliding window enables multiple threads to process the clique-clique overlap matrix as if it were in the memory in its entirely, while actually only a *chunk* physically resides in the memory. This is accomplished through an API which, on the one hand implements a mechanism necessary to effectively maintain in the memory only a portion of the matrix at any time. On the other hand it abstracts the former mechanism through simple functions.

It is a *sliding window* because a fixed size window is slid across the matrix, from the beginning down to the end. 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. $i \in [s, e] = \{j \in \mathbb{N} | s \leq j \leq e\}$. 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 $l \times l$ clique-clique overlap matrix; if the rows have indices between 0 and $(l - 1)$, we can solve the following equation:

$$\sum_{j=s}^{j=x} (l - (j + 1)) = \eta. \tag{3.1}$$

After some straightforward algebra, taking into account that $\sum_{i=1}^{i=n} i = n(n + 1)2^{-1}$ for each positive integer n, we find that the (3.1) has two solutions for x:

$$x_1 = \frac{2l - 3}{2} - \frac{1}{2}\sqrt{\Delta} \quad \text{and} \quad x_2 = \frac{2l - 3}{2} + \frac{1}{2}\sqrt{\Delta}.$$

Where

$$\Delta = (2s - 2l + 1)^2 - 8\eta.$$

It can be shown that (3.1) 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 (3.1) and considering 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 $w\left(\sum_{h=s}^{h=i}[l-(h+1)] + j - i - 1\right)$ to the first address of the window.

Parallel Chunk Processing. Each chunk is processed by two pools of threads.

First, a pool $\mathrm{T_B} = \{\tau_0, \tau_1, \ldots, \tau_{b-1}\}$ 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. In other words: for each pair of indices i, j in the clique-clique overlap matrix, $i > j$ iff the size of the maximal clique corresponding to i is less than (greater than) or equal to the size of the one corresponding to j.

When each thread in $\mathrm{T_B}$ has finished processing its own rows, another pool $\mathrm{T_C} = \{\tau_0, \tau_1, \ldots, \tau_{c-1}\}$ 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 connected components. Once again, mutual exclusion mechanisms have been avoided. In fact, the number of threads that actually process a generic element i, j is variable: only the threads $\tau_t \in \mathrm{T_C}$ s.t. t is less than or equal to the size of the maximum cliques associated with the element have to process it.

3.0.3 The Method

In FLIP-CPM, a main execution flow deals with the sliding of the window across the clique-clique overlap matrix and manages the startup (termination) of the various parallel flows of execution. Since the method extracts

k-clique communities in parallel for each possible value of k, i.e. from 3 to k_{max}, c set union data structures are required.

The method is summarized in 1. Once the maximal cliques are extracted and sorted by decreasing size (lines 1-2), they are labeled (line 3) with integers from 0 to $(l-1)$: the lower the label the higher the size. Then, set union data structures are initialized (line 4). For each k between 3 and k_{max}, an initial set consisting of n_k singleton sets is created. Each singleton set of the n_k corresponds to a maximal clique: this association is maintained by assigning it the same label as the maximal clique. At this point the window is slid on the clique-clique matrix. Every time the window is slid on a new chunk, operations in the outer loop are executed (lines 6-30). In this outer loop two distinct parts can be identified, separated by the synchronization point in line 17, in which multiple threads executes in parallel as explained in 3.0.2. In the first part (lines 9-16) the overlap is computed and written in the window. Note that the condition in line 10, given i, is true for one and only one thread in the pool. In the second part (lines 20-27) the overlap is read and G_k connected components are incrementally kept updated, following the algorithm described in 3.0.1. The assumption in line 19 is only meant to simplify the explanation of the operations that follow and it is correct for any value of k between 3 and k_{max}.

Method 1 Fast Lightweight Parallel CPM

```
Input: a graph $G = (V, E)$
Output: $k$-clique communities, $3 \leq k \leq k_{max}$
1: < compute the maximal cliques >
2: < sort the maximal cliques by decreasing size >
3: < label the maximal cliques from 0 to $(l-1)$ >
4: < initialize set union data structures >
5: $s, e \leftarrow 0$
6: while $e < (l-1)$ do
7:    $e \leftarrow Slide(s)$
8:    < start a pool of $b$ threads $\mathrm{T_B} = \{\tau_0, \tau_1, \ldots, \tau_{b-1}\}$ >
9:    for all the $i$ s.t. $i \in [s, e]$ do
10:      if this is the thread $\tau_t$, s.t. $t = i \pmod b$ then
11:         for all the $j$ s.t. $i < j < l$ do
12:            $ov_{i,j} \leftarrow$ < overlap between maximal cliques $i$ and $j$ >
13:            $Write(i, j, ov_{i,j})$
14:         end for
15:      end if
16:   end for
17:   < wait until each thread $\tau_i \in \mathrm{T_B}$ has terminated >
18:   < start a pool of $c$ threads $\mathrm{T_C} = \{\tau_0, \tau_1, \ldots, \tau_{c-1}\}$ >
19:   /* assume this is the thread $\tau_k$ */
20:   for all the $i$ s.t. $i \in [s, e]$ and $i < n_k$ do
21:      for all the $j$ s.t. $i < j < n_k$ do
22:         $ov_{i,j} \leftarrow Read(i, j)$
23:         if $ov_{i,j} \geq (k-1)$ then
24:            < check and possibly update $G_k$ connected components >
25:         end if
26:      end for
27:   end for
28:   < wait until each thread $\tau_i \in \mathrm{T_C}$ has terminated >
29:   $s \leftarrow e + 1$
30: end while
31: < analyse $G_k$ connected components, $3 \leq k \leq k_{max}$ >
```

When all the chunks have been processed the label of each maximal clique is searched for among the set union data structures (line 31). This is done only in structures that originally contained the corresponding singleton set, i.e. the ones used to keep the G_k connected components incrementally updated for ks between k_{max} and the size of the maximal clique.

A theoretical performance evaluation, at least for the first part of the outer loop (lines 9-16), can be done by analysing the speedup S. Let T_1 and p be the execution time of the sequential algorithm and the number of processors, respectively. Let ϵ be the difference between the time it takes for the slowest thread and the time it takes for the fastest one to compute and write overlap values for the rows in the window (i.e. operations between lines 9 and 16). Let $\mathcal{W}$ be the number of windows required to contain the entire clique-clique overlap matrix, roughly $l^2(2\eta)^{-1}$. The speedup S can be computed as:

$$S = \frac{T_1}{T_1/p + \epsilon\mathcal{W}} = \frac{p}{1 + p\mathcal{W}\epsilon/T_1}.$$

The lower the ϵ, the higher the speedup. In the limit $\epsilon \to 0$ the speedup equals the number of processors p and hence reaches the optimal case.

Chapter 4

Experimental Results

In this section we show the experimental results obtained by running an implementation of FLIP-CPM both on a standard personal computer and on a 48-core machine. When possible, we compare the execution times with those obtained by running an implementation of CPM, showing how the new method is more effective and efficient.

Our C implementation of FLIP-CPM uses the opensource development libraries of the package `igraph` [7] to extract maximal cliques. These libraries implement Bron-Kerbosh's algorithm. For parallel programming it uses the standard POSIX Threads[1], which provides an API for managing and handling threads. On the other hand, the implementation of CPM is available in CFinder [20], a free closed-source software tool. As input we used the Internet AS-level topology graph (built as described in [12]) and some IXP-induced subgraphs, i.e. tag-induced subgraphs [22] relating to ASs participating on Internet eXchange Points (IXPs). Figure 4.1(a) show the computation time experienced by running both FLIP-CPM and CPM on a standard dual-core personal computer, the iMac in Table 4.1(a). As inputs several small IXP-induced subgraphs were used (see Table 4.1(b) for their topological characteristics). In fact they are the only IXP-induced subgraphs on which CFinder executed without errors. It was not possible to run it on other subgraphs for the reasons discussed in Chapter 2. The new method was always at least one order of magnitude faster. This can partly be explained by the dual-core processor that enables FLIP-CPM of breaking down operations onto two parallel threads. Another reason that explains the huge reduction in execution time is certainly due to the implementations. In fact, CPM seems to be mostly implemented in Java and, although also it uses precompiled dynamic libraries, it is certainly less efficient than the new method, which is implemented entirely in C.

[1]POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)

(a) Machine Characteristics

	iMac	Dell
OS	Mac OS X 10.6.4	GNU/Linux
Kernel	Darwin 10.4.0	Linux 2.6.35.22
CPU	(2x)E7600 @ 3.06GHz	(48x)E7540 @ 2GHz
RAM	(4x)2GB @ 1067MHz	(16x)4GB @ 1067MHz

(b) Graphs Properties

	n	m	l	μ	σ^2
Internet	35,390	152,233	2,747,484	22.64	20.54
AMS-IX	322	13,434	752,108	22.20	9.55
LINX	345	14,188	384,494	23.01	11.29
DE-CIX	331	13,859	474,394	23.71	11.86
NL-IX	224	2,619	4,127	11.09	5.89
MSK-IX	293	4,225	1,593	12.36	9.42
SwissIX	116	1,110	669	8.62	2.46
MIX-IT	76	861	714	8.32	2.50
KleyReX	119	932	332	8.08	5.63

(c) Clique-Clique Overlap Matrix Density

Internet	0.976
AMS-IX	1
LINX	1
DE-CIX	1
NL-IX	0.976
MSK-IX	0.994
SwissIX	0.993
MIX-IT	0.939
KleyReX	0.968

Table 4.1: Hardware and software characteristics of the machines used for the experiments (a); properties of the graphs used (b) and the densities of the corresponding clique-clique overlap matrices (c). The density is the ratio between the number of elements with a value greater than zero and the total number of elements in the matrix. n, m and l are the numer of ASs (nodes), the number of BGP sessions (edges) and the number of maximal cliques respectively. $\mu = l^{-1} \sum_k k \cdot l_k$ is the average maximal clique size and $\sigma^2 = l^{-1} \sum_k l_k (k - \mu)^2$ the variance.

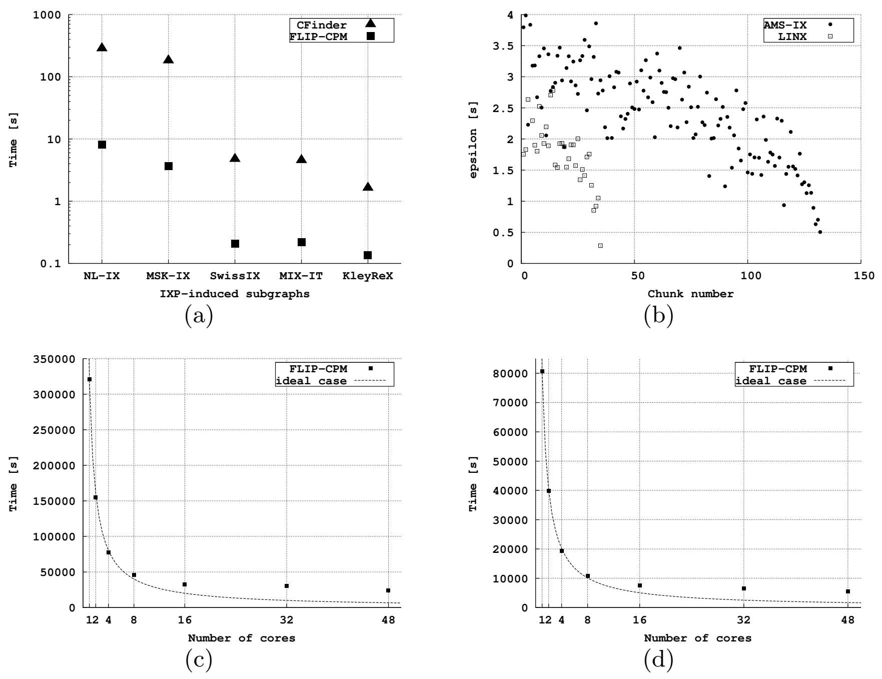
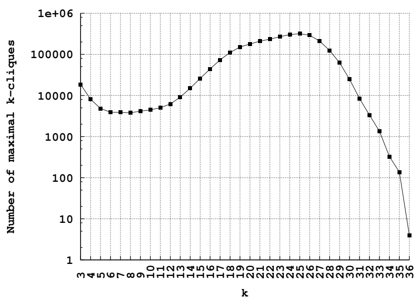


Figure 4.1: Computation times experienced by running FLIP-CPM and CPM on our iMac (a); ϵ experimented vs. chunk number (b) and computation time vs. number of cores on AMS-IX (c) and LINX (d), respectively. The round number is a counter that incremented by one every time the window is slid forward across the matrix.

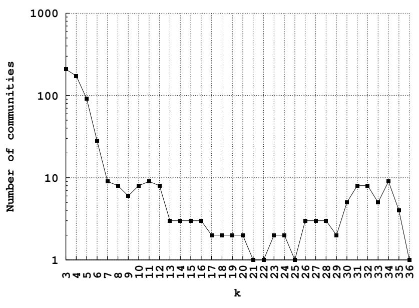
Figures 4.1(c) and 4.1(d) show the computation time experienced by running FLIP-CPM on an exponentially growing number of cores. The machine used is the 48-core Dell in Table 4.1(a). Two large IXP-induced subgraphs were used as inputs (see Table 4.1(b) for their topological characteristics). The dashed line in the figures represents the ideal case where doubling the number of cores implies halving the execution time (i.e. a speedup linear with the number of cores). With a number of cores 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 cores. When the number of cores becomes greater than or equal to sixteen, the speedup becomes less significant. The reduction in the speedup is due to the phase of the algorithm in which the pool of threads T_C reads the chunks and keeps the G_k connected components updated. Consequently, regardless of the number of cores available, the method uses a thread for each k and hence the maximum degree of parallelism is upper bounded by $k_{max} - 2$. Moreover, a lower bound on the execution time of this phase is due to the thread which has to carry out the highest number of operations (i.e. the thread which extracts 3-clique communities).

The ϵ, introduced in 3.0.3, is plotted in Figure 4.1(b) versus chunk number. The chunk number is just a counter that is incremented by one every time the window is slid forward across the matrix. Is evident from the figure a clear decreasing trend. This is caused by the fact that the rows become shorter and shorter as FLIP-CPM processes the lower parts of the clique-clique overlap matrix: the elements of the window that remain unused because of the floor operation become less and less. This implies an even better load balancing between threads. However, ϵ values are always less than 4 and 3 seconds on AMS-IX and LINX respectively and this shows how the load is actually well balanced between the various threads.

Finally, Figures 4.2(a) and 4.2(b) show two graphs relating to the Internet topology graph at the AS level. The first graph shows the number of maximal cliques versus k and helps to understand the complexity both of the Internet AS-level structure and of the problem we faced. The second graph show the number of k-clique communities versus k: a thorough analysis of these communities can be found in [13, 15]. The full extraction of the k-clique communities took about 23 days on the iMac and 3.84 days on the Dell.



(a)



(b)

Figure 4.2: Number of maximal k cliques (a) and number of k-clique communities (b) vs. k of the Internet AS-level topology graph

Chapter 5

Conclusions and Future Work

In this paper we tackled the problem of extracting k-clique communities from the Internet AS-level topology graph. We carried out a theoretical analysis of Clique Percolation Method (CPM), which revealed that it has significant scalability issues. These issues are both due to a quadratic dependence on the number of maximal cliques in the graph and make CPM extremely memory- and computation-absorbing when run on networks with a slightly higher number of such cliques. We have described Fast Lightweight 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 shared-memory computing architecture. With far fewer stringent requirements in terms of memory, the new method is shown to be experimentally able to extract k-clique communities 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.

Bibliography

[1] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. k-core decomposition: a tool for the visualization of large scale networks. *CoRR*, abs/cs/0504107, 2005.

[2] José Ignacio Alvarez-hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. k-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media*, 3(2):371–393, 2008.

[3] Zhaojun Bai, James W. Demmel, Jack J. Dongarra, Axel Ruhe, and Henk A. Van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* 2000.

[4] Vladimir Batagelj and Matjaz Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[5] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.

[6] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[7] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[8] Nan Du, Bin Wu, Liutong Xu, Bai Wang, and Xin Pei. A parallel algorithm for enumerating all maximal cliques in complex network. *Data Mining Workshops, International Conference on*, 0:320–324, 2006.

[9] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook.* CRC Press, Inc., 1st edition, 1998.

[10] Martin G Everett and Stephen P Borgatti. Analyzing Clique Overlap. *Connections*, 21(1):49–61, 1998.

[11] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[12] Enrico Gregori, Alessandro Improta, Luciano Lenzini, and Chiara Orsini. The impact of IXPs on the AS-level topology structure of the Internet. *Computer Communications*, 34(1):68 – 82, 2011.

[13] Enrico Gregori, Luciano Lenzini, and Chiara Orsini. k-clique Communities in the Internet AS-level Topology Graph. Technical report, 2010.

[14] Enrico Gregori, Luciano Lenzini, and Chiara Orsini. k-dense Communities in the Internet AS-Level Topology. In *COMSNETS 2011: Proceeding of the Third International Conference on COMmunication Systems and NETworkS*, 2010.

[15] Enrico Gregori, Luciano Lenzini, and Chiara Orsini. k-clique Communities in the Internet AS-Level Topology Graph. In *Submitted to IFIP Networking 2011*, 2011.

[16] Jussi M. Kumpula, Mikko Kivelä, Kimmo Kaski, and Jari Saramäki. Sequential algorithm for fast clique percolation. *Phys. Rev. E*, 78(2):026109, Aug 2008.

[17] Andrea Lancichinetti, Mikko Kivela, Jari Saramaki, and Santo Fortunato. Characterizing the community structure of complex networks. *PloS One 5(8)*, 2010.

[18] J. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3:23–28, 1965.

[19] Sue Moon, Jinyoung You, Haewoon Kwak, Daniel Kim, and Hawoong Jeong. Understanding Topological Mesoscale Features in Community Mining. In *COMSNETS 2010: Proceeding of the Second International Conference on COMmunication Systems and NETworkS (invited paper)*, pages 1 –10, 2010.

[20] Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[21] Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society - Supplementary Information, 2005.

[22] Gergely Palla, Ills J Farkas, Pter Pollner, Imre Dernyi, and Tams Vicsek. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, 10(12):123026, 2008.

[23] Kazumi Saito, Takeshi Yamada, and Kazuhiro Kazama. Extracting Communities from Complex Networks by the k-Dense Method. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E91-A(11):3304–3311, 2008.

[24] Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel Distrib. Comput.*, 69:417–428, April 2009.

[25] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31:245–281, March 1984.

[26] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28 – 42, 2006.

[27] Yun Zhang, Faisal N. Abu-Khzam, Nicole E. Baldwin, Elissa J. Chesler, Michael A. Langston, and Nagiza F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, 2005.