D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes

Alessio Conte
National Institute of Informatics, Japan
conte@nii.ac.jp

Tiziano De Matteis
University of Pisa
dematteis@di.unipi.it

Daniele De Sensi
University of Pisa
desensi@di.unipi.it

Roberto Grossi
University of Pisa
grossi@di.unipi.it

Andrea Marino
University of Pisa
marino@di.unipi.it

Luca Versari
University of Pisa
luca.versari@di.unipi.it

ABSTRACT

This paper studies k-plexes, a well known pseudo-clique model for network communities. In a k-plex, each node can miss at most $k-1$ links. Our goal is to detect large communities in today's real-world graphs which can have hundreds of millions of edges. While many have tried, this task has been elusive so far due to its computationally challenging nature: k-plexes and other pseudo-cliques are harder to find and more numerous than cliques, a well known hard problem. We present D2K, which is the first algorithm able to find large k-plexes of very large graphs in just a few minutes. The good performance of our algorithm follows from a combination of graph-theoretical concepts, careful algorithm engineering and a high-performance implementation. In particular, we exploit the low degeneracy of real-world graphs, and the fact that large enough k-plexes have diameter 2. We validate a sequential and a parallel/distributed implementation of D2K on real graphs with up to half a billion edges.

KEYWORDS

k-plexes, graph enumeration, community discovery, parallel programming

ACM Reference Format:
Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3220093

1 INTRODUCTION

Finding communities and clusters is one of the most fundamental tasks when analyzing any form of data, albeit computationally demanding. In networks, communities are generally associated with densely interconnected subgraphs [13, 21, 23]: a clique represents the ideal situation, where nodes are pairwise linked, and is the earliest and arguably most studied community model. Many modern approaches exist to find cliques in networks, based on the seminal paper by Bron and Kerbosch [5]. Great effort has been dedicated to optimize various goals, such as worst-case running time on general graphs [20] and sparse, real-world, graphs [12], usage of main memory [7], and running time as a function of the output [10].

In real networks, where data can be noisy or faulty, large and closely linked communities hardly appear as ideal cliques, so other forms of subgraphs are sought for. A natural answer to this question is a relaxed notion of *pseudo-clique*, such as k-core, k-plex, n-clan, n-club, s-clique, dense subgraph [13, 17, 19, 23], but there is another side to this coin: the number of pseudo-cliques grows exponentially, at an even faster pace than that of cliques; moreover, the cost of detecting the former ones is higher due to their more complex structure. These issues are well known and investigated in the literature.

In this paper, we focus on k-plexes, a widely used model of pseudo-cliques [3, 9, 19, 24, 25]: the requirement of each node being linked to all others is loosened to each node missing no more than $k-1$ links or, equivalently, missing k links including the one to itself (absence of self-loops is assumed). Some examples are shown in Figure 1. A clique is a 1-plex according to this definition, and as k grows, the number of k-plexes increases exponentially with respect to that of cliques: Figure 2 testifies how striking this is even on small networks (see Table 1 for reference). To make k-plexes effective models of communities, we want to focus on interesting configurations that are larger and more densely connected by fulfilling three constraints (trivially satisfied by cliques when $k = 1$): the first two appeared in the literature, and the last one is introduced and motivated here.

- The k-plexes should be *connected*, otherwise they cannot be regarded as one community.
- Their size should be *large*, so as to involve as many entities as possible. In particular, they should be *maximal* under inclusion, namely, adding one more node violates the constraints given here.
- Their *diameter* (i.e. maximum pairwise distance among their nodes) should be *small*, as a node cannot go too far in reaching the nodes in its community.

In particular, we require that the diameter should be at most 2 for the following reasons: on one hand, it is equivalent to saying that every two nodes in the community are directly linked or at least

KDD '18, August 19–23, 2018, London, United Kingdom

ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3220093

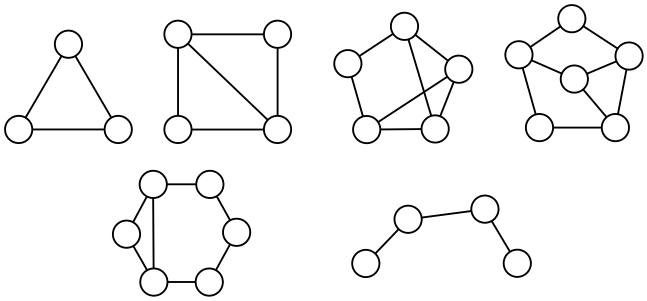

Figure 1: From left to right, top: a 1-, 2-, 3-, 4-plex. Bottom: a 4-plex and a 3-plex with diameter larger than 2.

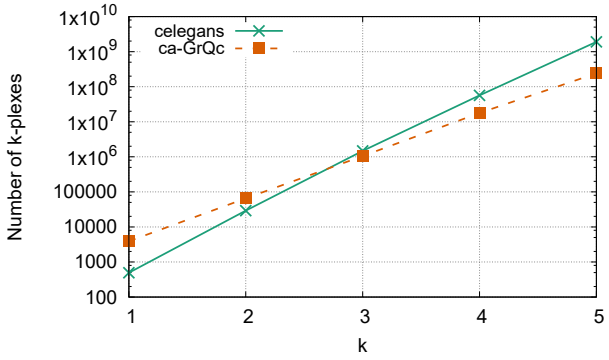

Figure 2: Number of k-plexes for increasing values of k.

share a common neighbor. Indeed, any k-plex with diameter greater than 2 has nodes that are not linked and do not share common neighbors: it could be argued that such nodes should not belong to the same community (see the examples in Figure 1). On the other hand, if the diameter is less than 2, we obtain exactly a clique. Asking the diameter to be *at most 2* is the right balance between these extremes. Furthermore, this does not miss any significant communities (the meaning of significance is discussed in Section 6) in comparison with standard k-plexes. Specifically, *all* connected 1-plexes and 2-plexes have diameter at most 2, and the only connected 3-plex of diameter more than 2 is the induced path of 4 nodes, which can hardly be considered a community. For larger values ($k \geq 4$), we still guarantee to find at least all k-plexes with $2k-1$ or more nodes, as it is well known that they have diameter 2 (e.g., see Lemma 5.1 in [9]). As k is a small constant in practice, this guarantees to hit all large (or even medium-sized) communities.

From a computational point of view, the fact that the diameter is at most 2 gives us a powerful handle to efficiently pull out the most interesting k-plexes from massive networks. A practical example can be seen in Section 2, where we briefly describe the biggest communities in some large real networks, whose detection only took few minutes in our case.

Our main contribution is the algorithm D2K (for diameter-2 k-plexes), which shows that significant k-plexes can be efficiently discovered in large networks. D2K features a recursive backtracking structure, and a decomposition which is particularly suitable for parallelization and distribution. Furthermore, we introduce some novel pruning techniques for the problem, based on simple yet effective insights on both the problem and the structure of real data, which dramatically reduce the search space and consequently the computation time.

While computation of cliques on large and sparse real-world networks is feasible with ad-hoc algorithms[10, 12], computing k-plexes on such networks has been so far a task out of reach: [9] finds the largest 2-plex on a graph with 1.8 million edges, and [24] scales up to a graph with 22 million edges (see *pokec* in Table 1), but only processes a subgraph made by 10 of its nodes and the surrounding areas.

While these results greatly improve upon previous approaches (e.g., [3, 25]), it is not yet enough for today's data: large real world graphs can have hundreds of millions of nodes, or even billions. We argue that our approach can give a satisfying solution to the sought-after problem of finding pseudo-cliques in very large graphs. This is motivated in Section 5, where we show how our approach can process real world graphs with hundreds of millions of nodes in very reasonable times. Moreover, further discussion with related work is given in Section 6.

2 DATA ANALYSIS

For the sake of presentation, we computed the largest 4-plexes on three networks, whose content is briefly described below. Although performing data analysis is out of the scope of this paper, we believe that the following toy examples can help to grasp its flavour.

`Pokec`. This dataset is a snapshot of the most popular Slovakian social network. The largest 4-plex is a group of 32 users, aged between 15 and 20 years. Remarkably, the community is mainly Czech, with 81% of the users from Czech Republic (whose language, should be remarked, is closely related to Slovak), and is 94% female. Furthermore, data suggests it may have been a pre-existing group of friends who decided to join together, as 97% registered within 5 months of each other (January to May 2012). The users also seem to have similar music interests, with 56% selecting Slovak actress and singer *Lucia Molnárová* as favorite singer.

`it-2004`. This is a 2004 crawl of the Italian web (*.it* domain) made by [4]. The largest 4-plex consists in a collection of 3210 pages from the website *www.cuoko.it*. The reason behind such a structure is not apparent from the data, probably an attempt for search engine optimization or for increasing user activity.

`uk-2005`. This is a 2005 crawl of the British web (*.co.uk* domain, but including *bbc.com*) again made by [4]. The largest 4-plex consists in a collection of 587 pages from a series of similarly named housing websites, plus the two domains *gibbinsrichards.co.uk* and *doorkeys.co.uk*. The following is an excerpt.

gibbinsrichards.co.uk	homes-for-sale-oxford.co.uk
doorkeys.co.uk	homes-for-sale-paisley.co.uk
…	…
estate-agent-oldham.co.uk	house-for-sale-paisley.co.uk
estate-agent-oxbridge.co.uk	house-for-sale-perth.co.uk
…	…
home-for-sale-oxbridge.co.uk	houses-for-sale-perth.co.uk
home-for-sale-oxford.co.uk	houses-for-sale-peterborough.co.uk
…	…

Among these pages, only *gibbinsrichards.co.uk* is currently active and indeed corresponds to a housing company. While a *whois* lookup does not reveal any information on the similarly named websites, one may conjecture that they were bogus websites set up for the benefit of the first two domains, e.g., for improving their rank scores on well known search engines.

3 ALGORITHM

In this section we show our proposed algorithm for listing maximal k-plexes of diameter at most 2 for a given integer $k \geq 1$, on a graph $G = (V, E)$. Each listed solution is a set $K \subseteq V$, such that the resulting induced subgraph $G[K]$ has the following properties: each node has degree at least $|K| - k$ (i.e., it is a k-plex), it is connected, and the maximum pairwise distance among its nodes is ≤ 2. All solutions are also maximal under inclusion, i.e., there is no $K' \supset K$ satisfying the latter properties. Moreover, an optional threshold $q > 0$ can be given, so that $|K| \geq q$ is guaranteed in order to discover large k-plexes.

First we will explain the core structure of our approach, then some crucial algorithm engineering and optimization which allow us to reduce the search space.

3.1 Main structure

We assume the nodes to be ordered arbitrarily as $v_1, \ldots, v_n$. We say that v_i is smaller than v_j if it comes earlier in the ordering, i.e., $i < j$. For a node v_i, let $N(v_i)$ be the set of nodes adjacent to v_i (its neighbors), and $N_>(v_i)$ be the set of *forward neighbors* of v_i, i.e., $N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$. Furthermore, let $N^2_>(v_i)$ be the set of *forward cousins* of v_i, that are neighbors of a node in $N_>(v_i)$ which are also larger than v_i (and not already forward neighbors of v_i).

We decompose the original problem into enumerating, for each node v, all k-plexes whose smallest node is v itself. To do so, we will exploit the following key property:

Observation 3.1. *When v_i is the smallest node in a diameter two k-plex, all other nodes of the k-plex must be either forward neighbors or forward cousins of v_i.*

This allows us to look for solutions with smallest node v in the subgraph containing just v, its forward neighbors, and its forward cousins, which we call BLOCK(v). To improve the performance of our approach, we want BLOCK(v) to be as small as possible, as will be discussed later.

In order to process each BLOCK(v), we use a binary partition scheme similar to the Bron-Kerbosch clique enumeration algorithm [5]. We want to recursively solve the following subproblem.

Problem 1. *Given a connected k-plex K and a set of nodes* EXCL, *find all maximal connected k-plexes K' such that $K \subseteq K'$ and $K' \cap$ EXCL $= \emptyset$.*

Solving Problem 1 recursively is easy: for a given $v \notin K \cup$ EXCL such that $K \cup \{v\}$ is a connected k-plex, we split the problem into finding all solutions containing v by adding v to K, and all those who do not by adding it to EXCL instead, using two nested recursive calls. For efficiency, for each k-plex K we keep a set CAND of all v that should be tested, and directly generate all nested calls that would add one node of CAND to K and the previously considered one to EXCL as children of the same recursive call.

Whenever CAND is empty, it is easy to see that K is a maximal connected k-plex if and only if EXCL is also empty, since any node left in EXCL may be added to K to make a larger k-plex, and if both sets are empty no node can be used to enlarge K.

Finally, for any k-plex maximal in BLOCK(v), our algorithm checks whether it is maximal in G, in which case it is output, or it is a subset of a larger k-plex whose smallest node is some $w < v$, in which case it is discarded as it is not a solution.

The resulting algorithm is as follows:

Algorithm 1: Structure of our algorithm

```
Input  : A graph G = (V(G), E(G)), an integer k
Output : All diameter 2 k-plexes of G
1  foreach v in an ordering {v_1, ..., v_n} of V(G) do
2      H = (V(H), E(H)) ← BLOCK(v)
3      enum(H, {v}, N(v) ∩ V(H), ∅)
4  Function enum(H, K, CAND, EXCL)
5      if CAND ∪ EXCL = ∅ /*K is maximal in H*/ then
6          if K is maximal in G then
7              output K
8      foreach c ∈ CAND do
9          CAND', EXCL' ← update(H, K ∪ {c}, EXCL)
10         enum(H, K ∪ {c}, CAND', EXCL')
11         CAND ← CAND \ {c}
12         EXCL ← EXCL ∪ {c}
13 Function update(H, K, EXCL)
14     CAND' ← {v ∈ V(H) \ (K ∪ EXCL) : K ∪ {v} is a connected
         k-plex }
15     EXCL' ← {v ∈ EXCL : K ∪ {v} is a connected k-plex }
```

For each child, CAND and EXCL are updated using the function `update()` to retain just the nodes x such that $K \cup \{x\}$ is still a connected k-plex.

To speed up the `update()`, we can compute CAND' and EXCL' by difference from the current ones: the addition of c to K may only shrink EXCL. As for CAND, we may have to add some nodes which became connected to K, namely, neighbors x of c which had no neighbor in K. Whenever $|K| \geq k$, however, our algorithm can skip this check, as any such x would have at least k non-neighbors in K, meaning that $K \cup \{x\} \cup \{c\}$ would not be a k-plex.

A strong point of this algorithm is the suitability for parallel and distributed computation, since each BLOCK(v) can be processed independently. Furthermore, we will now present how the computation on each block can be further optimized.

3.2 Degeneracy ordering

The first and most surprising cut is simply ordering $V(G)$ in a *degeneracy ordering* [12].

If we call Δ the maximum degree of a node in G, note that the number of nodes in BLOCK(v) is bounded by Δ^2. However, a degeneracy ordering minimizes the maximum number of *forward* neighbors of a node in G: this number is called the *degeneracy* of

the graph, d. Using such ordering reduces the size of $\textsc{block}(v)$ from Δ^2 to $d \cdot \Delta$

Extensive experimental evidence has shown the degeneracy to be a small number on most real-world networks, even large ones [10–12], and can be noticed in Table 1. The practical effect of this ordering is striking. Horizontal lines *baseline* and *d* in Figure 3 show the maximum block sizes produced by our algorithm, using respectively a *random* or a *degeneracy* node ordering: the latter ordering produced blocks always less than half the size of the former, and up to a factor 7. (Lines *sp* and *sp* + *d* will be covered later in Section 3.4)

3.3 Pivoting generalization

The Bron-Kerbosch algorithm for clique enumeration uses an effective technique called *pivoting* to cut useless branches from the computational trees. This is based on the principle that "each maximal clique must contain either the node u or a non-neighbor of u". In this paper we present of a generalization of this principle, applied to k-plexes:

Lemma 3.2 (k-plex pivoting). *Let K be a k-plex, $\textsc{cand} = \{v \notin K : K \cup \{v\}$ is a k-plex$\}$, and u a node in $\textsc{cand}$. Any maximal k-plex containing K contains either u, a non-neighbor of u, or a neighbor v of u such that v and u have a common non-neighbor in K.*

Proof. Assume by contradiction that K' is a maximal k-plex that violates the above constraint: u has at most $k-1$ non-neighbors in K' since $K \cup \{u\}$ is a k-plex and $K' \setminus K$ is made by only neighbors of u. Furthermore, any non-neighbor w of u in K is a neighbor of all nodes in $K' \setminus K$, thus w may not have more than $k-1$ non-neighbors in K' as it did not in K. Thus $K' \cup \{u\}$ is a k-plex and K' is not maximal, a contradiction. □

We say that this is a generalization of the pivoting for clique enumeration as for $k = 1$ we obtain exactly the pivoting of the Bron-Kerbosch algorithm. Furthermore, as for the Bron-Kerbosch algorithm, the same applies if u is chosen in $\textsc{excl}$ rather than $\textsc{cand}$. The cut of the search space is in that, when considering K, we can skip the recursive call for nodes which are both neighbors of u *and* do not share any non-neighbors in K with u. In practice, this translates to replacing Line 8 in Algorithm 1 with the following

foreach $\{c \in \textsc{cand} : c \notin N(u)$ or $K \setminus (N(u) \cup N(c)) \neq \emptyset\}$ **do**

where u is the chosen pivot. In order to maximize the effectiveness of this cut, we adopt the philosophy of Tomita et al. [20], and pick at each step the u which maximizes the number of prevented recursive calls.

3.4 Solution size pruning

Finally, we show some cuts that leverage our interest in k-plexes with minimum size q.

The first and most obvious cut follows from the fact that all the k-plexes generated from a given recursive call will be a subset of $K \cup \textsc{cand}$. This means that, whenever $|K \cup \textsc{cand}|$ is less than q, we can cut the search as no interesting solutions will be produced.

Another less obvious, yet essential cut, is obtained from the following lemma:

Lemma 3.3 (size pruning). *Any two nodes u and v in a k-plex K of size q, have at least $q - 2k + 2$ common neighbors in K.*[1]

Proof. As each node has at most $k-1$ non-neighbors in K, the number of node in K that are not neighbors of at least one of u and v is at most $2k-2$. From this follows that there are at least $q - 2k + 2$ nodes that have both of them as neighbors. □

As when processing $\textsc{block}(v)$ we are only interested in k-plexes of size at least q containing v, this means we can immediately (and recursively) remove from $\textsc{block}(v)$ any node that does not share $q - 2k + 2$ neighbors with v.

The effect of this cut is not just heuristic: we give no proof for space reasons, but simple calculations show that this reduces the maximum size of $\textsc{block}(v)$ from $d \cdot \Delta$ to $d \cdot \Delta/(q - 2k + 2)$.

In practice, as q grows, size pruning may reduce the size of the subgraphs processed by up to orders of magnitude, as shown in Figure 3: line *sp* shows the size maximum size of a block generated on the corresponding graph with the specified q, using the size pruning. Compared to the *baseline* (i.e., no optimization) one can see how the difference is already important for $q = 4$, and becomes even larger for increasing q, so much so that for $q = 30$ on most graph there is hardly anything left to process. Finally, line *sp* + *d* in Figure 3 represent the maximum block size generated by D2K, i.e., using both the degeneracy ordering and the size pruning: we can see how this technique takes the best of both reductions, obtaining much smaller sizes than the baseline for all values of q. Notably, the effectiveness of the cuts seems to be independent from the size of the graph, but strongly influenced by the degeneracy, as we can observe a greater reduction on email-euall and pokec rather than on ca-grqc and in-2004.

These cuts allow us to process even huge graphs in a short time, provided a large enough q is chosen (see Section 5).

4 PARALLELIZATION

Concerning the parallelization of the algorithm, we first describe how we parallelized it for the execution on a single shared memory multi-core machine. Then, we extend our parallelization to support a cluster of multi-core computing machines with a distributed memory.

4.1 Shared Memory, Single Machine

The parallel implementation of D2K is characterized by the presence of multiple threads, each one performing part of the processing. For doing this, we resort to the observation that each $\textsc{block}(v)$ can be processed independently from the others (see Section 3.1). Accordingly, the first step of our solution is to keep all the nodes v in a queue shared among the threads (in the following *vqueue*).

Each thread extracts a node from the queue and begins to process the BLOCK associated with that node. When a thread terminates the processing of a BLOCK, a new node is extracted from *vqueue*, until there are no more nodes (and thus BLOCKs to be processed). Since the time required to process different BLOCKs may be different, with this solution we could experience some load unbalancing effects. In particular, it could happen that most threads have terminated the available BLOCKs while few threads are still processing some heavier

[1] If $u \in N(v)$, we consider u as common neighbor of u and v

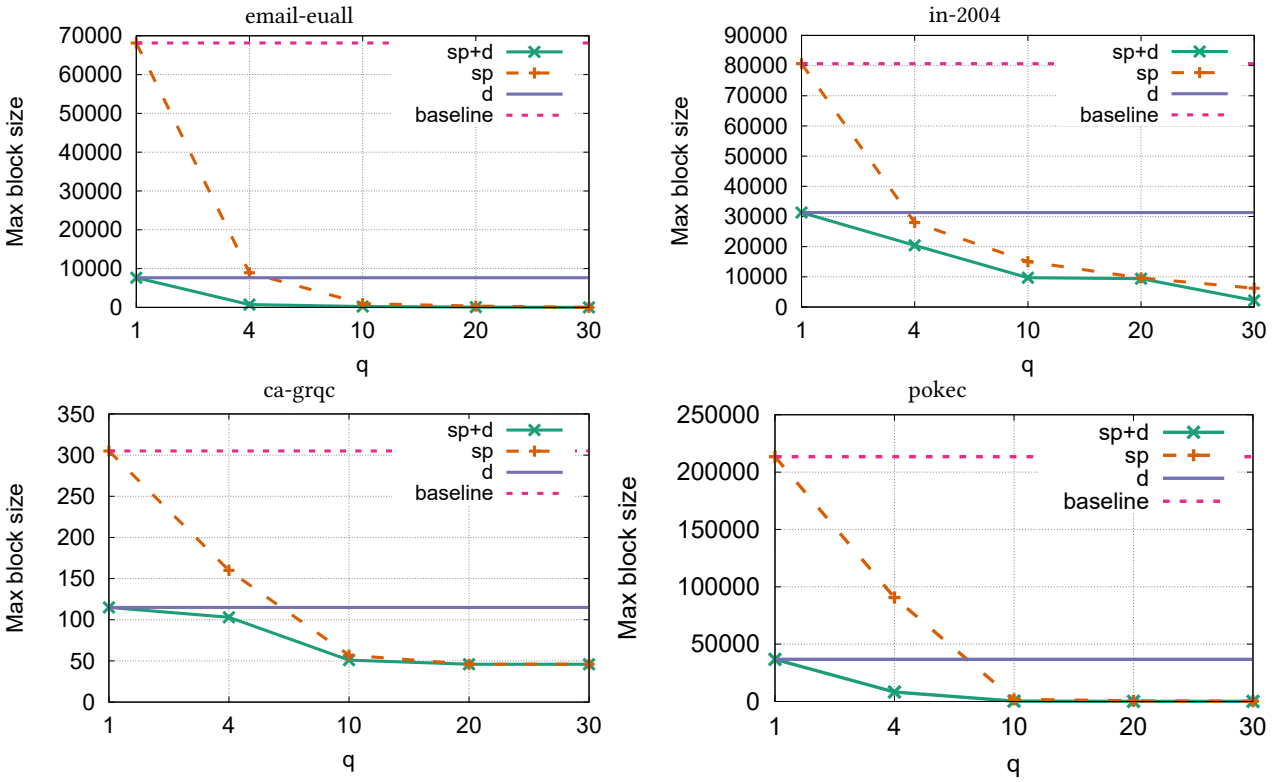

Figure 3: Effectiveness of the pruning techniques described in Sections 3.2 and 3.4 on the size of the generated BLOCKS

BLOCKS. To mitigate this effect, we use a *work requesting* approach. If a thread detects that there are no more BLOCKS to be processed but some thread is still processing something, it will require the offloading of additional work to improve the load balancing and reduce the execution time.

The offloading is performed by means of a globally shared queue *wqueue*. We keep an atomic counter indicating how many threads have finished all the BLOCKS and are available for additional work. During the processing of a BLOCK, each thread periodically checks if the size of *wqueue* is lower than the value of the counter. If this is the case, the thread will put a part of the BLOCK in the queue, instead of processing it directly. The requesting worker will then be able to dequeue the sub-BLOCK from *wqueue* and to process it. To implement this, D2K can generate an object representing a nested recursive call (i.e., the sub-BLOCK), then hand it over to the requesting worker, who will process it, and skip directly to the next recursive call.

4.2 Distributed Memory, Multiple Machines

We adopt a similar approach for a cluster. Since we are in a distributed memory environment, we adopt a *master-worker* solution. One computing machine will act as the *master*, dealing with the distribution of the BLOCKS to be processed to a set of *workers*, each one executed on a different multi-core computing machine.

When the computation starts, the master sends a chunk of nodes to each worker. A generic worker, after receiving a chunk, will insert all the nodes in its shared queue, leveraging on the parallelization scheme used for the single computing machine to process all the assigned nodes. Differently from the single machine case, when there are no more BLOCKS to be processed, instead of terminating the execution, the worker will ask the master for a new chunk.

When the master receives a request for a new BLOCK from a worker w_{req} and there are no more BLOCKS to assign, the master tries to balance the work by requesting a BLOCK (or a part of it) from an overloaded worker w_{victim} and by redirecting it to w_{req}. Each worker periodically checks for pending stealing requests from the master. When this is the case, and there are other nodes in the current chunk which were not yet processed, the worker sends some of those nodes to the master; otherwise (i.e., it is processing the last node of the assigned chunk), it sends a sub-BLOCK to the master. In both cases, the master will redirect the work received from w_{victim} to v_{req}. This process continues until all the workers have nothing left compute.

To select w_{victim} we adopt a simple heuristic. We store the timestamp of the last chunk request received by each worker. When selecting a victim for the stealing, we pick the worker characterized by the lower timestamp, since the worker who has not requested data for the longest time is likely to be the most loaded worker, i.e., the most suitable victim for stealing.

Network	Type	n	m	Δ	d	2-plexes			3-plexes	
						$q = 4$	$q = 10$	$q = 20$	$q = 10$	$q = 20$
caida	as	26 475	53 381	2 628	22	1 337 044	23 314	0	1 531 876	0
ca-grqc	collab	5 241	14 484	81	43	12 038	377	118	13 352	1 568
jazz	collab	198	2 742	100	29	26 172	8 059	2	257 233	2
celegans	bio	354	1 501	186	10	12 814	5	0	248	0
homo-sapiens	bio	1 027	1 166	443	17	929 795	576	0	20 301	0

(a) Small Networks

Network	Type	n	m	Δ	d	2-plexes			3-plexes	
						$q = 4$	$q = 10$	$q = 20$	$q = 10$	$q = 20$
interdom	bio	1 706	78 983	728	129	60 858 742	54 562 092	34 424 664	536 552 584 954	378 852 643 230
amazon0505	prod	410 236	2 439 436	2 760	10	8 996 909	22 483	0	248 433	0
email-euAll	comm	265 214	365 569	7 636	37	11 489 357	1 042 929	0	75 751 394	2 637
epinions1	soc	75 888	405 739	3 044	67	176 891 842	73 518 387	3 286 364	11 947 340 789	538 275 804
slashdot090221	soc	82 144	500 480	2 548	54	62 130 214	28 862 926	11 411 028	2 993 676 468	1 303 148 522
wiki-vote	soc	8 298	100 761	1 065	53	42 757 442	9 162 660	52	1 337 634 391	156 727

(b) Medium Networks

Network	Type	n	m	Δ	d	2-plexes		3-plexes	
						$q = 50$	$q = 100$	$q = 50$	$q = 100$
as-skitter	as	1 696 415	11 095 298	35 455	111	47 969 775	0	21 070 497 438	0
in-2004	web	1 353 703	13 126 172	21 869	488	25 855 779	9 978 037	29 045 783 792	4 257 410 159
						$q = 20$	$q = 30$	$q = 20$	$q = 30$
pokec	soc	1 632 803	22 301 964	14 854	47	94 184	3	5 911 456	5
						$q = 250$	$q = 500$	$q = 250$	$q = 500$
uk-2005	web	36 022 725	549 996 068	1 372 171	584	106 243 475	256 406	$\geq$ 18 336 111 409	28 199 814
						$q = 2000$	$q = 3000$	$q = 2000$	$q = 3000$
it-2004	web	35 051 939	473 649 719	1 243 927	3 209	2 727 030	6 304	$\geq$ 165 823 718	2 722 875

(c) Big Networks

Table 1: Considered networks and their properties, including number of k-plexes of different sizes

5 EXPERIMENTS

In this section we show an experimental evaluation of d2k, and a comparison with the fastest known algorithms from [9] and [24] for listing large k-plexes. As all algorithms produce the same output (k-plexes of size at least q), we only compare the performance. We remark that this paper only deals with the problem of efficiently listing k-plexes. A qualitative study on the difference between k-plex communities and other types of pseudo-cliques would be interesting to investigate, but is out of the scope of this paper.

The evaluations have been performed on a cluster of 16 homogeneous machines. Each machine features a dual CPU Intel Xeon E5-2640 v4, Broadwell based architecture, composed of 20 cores operating at 2.40GHz and 128 GB of RAM. Each core has a private L1 (32KB) and L2 (256KB) cache, and each CPU is equipped with a shared L3 cache of 25MB. The HyperThreading feature was not used. The machines are interconnected using a OmniPath network.

The program is written in `C++11`, compiled with `gcc-6.4.0`, using the `-O3` optimization flag. The source code of our algorithm is publicly available at https://github.com/veluca93/parallel_enum. In our implementation we used `C++11` threads for the shared memory parallel implementation, and `MPI` plus `C++11` threads for the distributed implementation. In the following discussion, the sequential and parallel evaluations are performed on a single machine, while the distributed tests use a different number of machines. For all tests, we considered 12 hours of execution time as a hard limit.

Dataset. The graphs considered in our experiments are reported in Table 1. They correspond to graphs taken from SNAP (http://snap.stanford.edu/) and portions of web-crawls taken from LAW (http://law.di.unimi.it/). We divided our networks by size in three classes: the first class, shown in Table 1(a), contains networks with up to (approximately) 50 thousand edges, the second one (b) contains the remaining networks with less than 10 millions edges, while the larger graphs are in the third class (c). We use classes (a) and (b) to compare our approach with the existing ones, setting $q = 4, 10, 20$ for the 2-plexes and $q = 10, 20$ for the 3-plexes, where q is the minimum solution size requested. We then use the bigger networks to study the behaviour of our approach in parallel and distributed settings. Since these networks are much bigger, and contain many more k-plexes, we change the value of q from network to network. We will later show how to choose suitable values for q. For each network we report some statistics and its type.[2] In particular, we report the number of 2-plexes and 3-plexes greater than q for different values of q. For some of these values, we show just a lower bound on this number (the ones reporting $\geq$) since our computation exceeded the time limit of 12 hours. This happened in the cases where there was a huge number of solutions.

It is worth observing that graphs with a similar number nodes or edges can contain a very different number of k-plexes. This may

[2] as: autonomous systems, collab: collaboration, bio: biological, prod: product co-purchasing, comm: communication, soc: social, web: web crawl.

be related to a higher density, and it seems that a high average degree can correspond to a huge amount of 2- and 3-plexes. The most striking example is the biological graph `interdom`, relatively small but with average degree 92.6, that has millions of 2-plexes and billions of 3-plexes. On the other hand, the collaboration network `ca-grqc` has a similar number of nodes and a smaller average degree (about $1/17th$ that of `interdom`), and indeed the number of 2- and 3-plexes is orders of magnitude smaller. In some other cases, the number of k-plexes is more deeply related to the topology of the networks. This is the case of `as-skitter` and `in-2004` which have roughly the same number of nodes and edges but the number of 2-plexes greater than 100 is very different: almost 10 millions for `in-2004` and 0 for `as-skitter`. Small networks with a huge amount of large k-plexes, like `interdom`, seem to be uncommon, meaning that despite the exponential upper bound, the number of k-plexes in real-word networks is in practice relatively small for suitable values of q.

Choosing q. Due to the effectiveness of the pruning techniques, D2K finishes quickly when q is too high to find any result. This feature can be exploited to quickly find the largest k-plexes in a graph. The philosophy is similar to that used in [9], but with the important difference that we do not require listing all maximal cliques first, something which is trivial on small graphs but far from it on larger ones. For the larger graphs in Table 1(c), we thus adaptively found values of q corresponding to their larger communities.

5.1 Performance Evaluation

In this section we evaluate the performance of our method D2K with respect to the competitors GP [24] and LP [9] using the graphs in Table 1(a) and (b) when generating 2-plexes and 3-plexes of size at least q. All the methods were run in a sequential setting. Moreover, note that, as q is always greater than k^2, all the k-plexes to be listed have diameter at most 2, which means that all the algorithms return the same set of solutions.

To give a general picture of the running times for the different values of q, we reported time performances of all the approaches in Table 2 using the small graphs in Table 1(a). It is worth observing that LP and GP show their best behaviour in opposite scenarios: LP runs out of time (12 hours) for smaller values of q and improves when q grows; GP is faster than LP for small values of q, but its performance does not improve as q grows, even when there are no solutions to list, i.e., no k-plexes of size q or larger (see the values reporting * in Table 2). Indeed, it seems that GP sometimes spends a lot of time in cases where there are no solutions, while for LP this does not seem to happen.

Our proposed algorithm D2K always outperforms both competitors, in most cases by at least an order of magnitude, terminating in less than one second or few seconds for all the values of q for both 2- and 3-plexes.

In order to quantify our improvement with respect to the state of art, we used the medium sized graphs in Table 1(b). We report the time needed by D2K (indicated as T) and the speedup S of D2K with respect to the best running time achieved by both LP and GP. In particular, we divide the best running time of the competitors by T to obtain S. Table 3 reports T and S for both 2- and 3-plexes, for different values of q. OOT means that D2K ran out of the maximum time allowed (12 hours). The speedup S is set to $\star$ whenever both our competitors ran out of the maximum time or memory allowed. For the great majority of the graphs the values of S is always greater than 20, meaning that we spent less than $1/20th$ of the running time of our competitors. In the case of the 2-plexes with $q = 10$ and $q = 20$, our improvement is even more evident, as D2K is much faster and often outperforms the competitors by orders of magnitude. In the case of 3-plexes with $q = 20$ and $q = 30$, for the great majority of the cases both our competitors ran out of time or memory while D2K finished in time. On the other hand, when setting $q = 10$, D2K and both the competitors went out of time. We will be able, however, to deal with these graphs using the parallel version of our algorithm. Finally, Table 3 (in particular with respect to 3-plexes) shows how the hardness of k-plex enumeration is linked to the density, rather than size, of the graphs. The graph `interdom`, the smallest in this table, is the hardest one to process, as even for $q = 30$ D2K (and the competitors) run out of time; on the other hand, it can be seen in Table 1 how `interdom` has the highest degeneracy among these graphs, meaning that it is denser. Similarly, `epinions1` has the second-highest degeneracy in this graph, and it is the only one other than `interdom` to run out of time for $q = 20$.

Setup time. Since D2K needs the degeneracy ordering of the nodes of the given network, we report for the sake of completeness the time needed to compute this ordering on the networks in our dataset. It can be observed that this setup time (shown below, in seconds) is completely negligible with respect to the time needed to list k-plexes (see Table 2 and Table 3).

Networks	Time	Networks	Time
ca-grqc	0.01	email-euAll	0.12
celegans	0.01	slashdot090221	0.13
jazz	0.01	amazon0505	0.73
caida	0.02	as-skitter	2.86
homo-sapiens	0.02	in-2004	1.95
interdom	0.02	pokec	5.45
wiki-vote	0.04	uk-2005	79.21
soc-epinions1	0.10	it-2004	94.04

5.2 Evaluation of the parallel and distributed implementations

For the parallel evaluation of D2K, we executed the program on a single machine of the cluster with a variable number of threads, ranging from 1 to 20, the number of cores available in the machine.

To evaluate the effectiveness of the parallelization, we consider the speedup obtained with respect to the sequential version, i.e. the ratio between the execution time of the sequential implementation over the execution time of the parallel version with a given number of threads. Figure 4 shows the speedup obtained by the parallel version for increasing number of threads on two scenarios (2-plexes of size at least 4, and 3-plexes of size at least 20) for three example graphs. In both situations D2K obtains good speedups, close to the ideal one (represented by the dotted line).

Table 4 reports the parallel execution time obtained with 20 threads and the corresponding speedup with respect to the sequential implementation for the graphs in Table 1(b). We obtained good

Network	2-plexes									3-plexes					
	$q = 4$			$q = 10$			$q = 20$			$q = 10$			$q = 20$		
	D2K	GP	LP	D2K	GP	LP	D2K	GP	LP	D2K	GP	LP	D2K	GP	LP
caida	**18.29**	449.89	OOT	**0.72**	357.35	OOT	*__0.01__	*321.49	*0.10	**50.86**	OOT	OOT	*__0.01__	*OOT	*OOT
ca-grqc	**0.07**	3.98	OOT	**0.01**	3.19	2 964.91	**0.01**	2.50	12.49	**0.07**	17.27	OOT	**0.01**	48.42	471.90
jazz	**0.25**	6.47	OOT	**0.12**	5.28	OOT	**0.01**	2.40	OOT	**2.89**	29.12	OOT	**0.01**	41.26	OOT
celegans	**0.08**	3.27	OOT	**0.01**	1.93	1 186.19	*__0.03__	*1.65	*0.04	**0.01**	48.20	OOT	*__0.01__	*50.48	*__0.01__
homo-sapiens	**4.89**	123.17	OOT	**0.02**	101.91	OOT	*__0.01__	*91.71	*0.10	**1.27**	OOT	OOT	*__0.01__	*24 383.34	*__0.01__

Table 2: Time comparison among D2K, GP [24], and LP [9] for generating all 2- and 3-plexes of size greater than q (time is in seconds) for the small graphs in Table 1(a). OOT: the process did not terminate within 12 hours. *: there were no solutions.

Network	2-plexes						3-plexes					
	$q = 4$		$q = 10$		$q = 20$		$q = 10$		$q = 20$		$q = 30$	
	T(s)	S	T(s)	S	T(s)	S	T(s)	S	T(s)	S	T(s)	S
interdom	1 032.47	16.24x	1 040.76	15.45x	989.02	14.39x	OOT	★	OOT	★	OOT	★
amazon0505	56.76	28.68x	0.33	4 183.13x	*0.01	600x	4.16	★	*0.01	2 081x	*0.01	556x
email-euAll	157.35	56.53x	31.46	228.09x	*0.65	9384x	2 995.51	★	76.96	★	2.99	★
epinions1	2 945.71	★	1 545.27	★	283.93	67.75x	OOT	★	OOT	★	3 720.38	★
slashdot090221	989.99	39.81x	408.99	88.74x	260.39	90.29x	OOT	★	30 278.48	★	3 313.04	★
wiki-vote	532.22	22.16x	262.72	27.01x	16.79	76.59x	OOT	★	2 763.42	★	341.17	★

Table 3: Time (T) of D2K and speedup (S) D2K obtains with respect to the fastest between GP and LP for medium sized graphs. OOT: D2K ran out of time. ★: both GP and LP ran out of time.

Network	2-plexes						3-plexes					
	$q = 4$		$q = 10$		$q = 20$		$q = 10$		$q = 20$		$q = 30$	
	T (s)	S	T (s)	S	T (s)	S	T (s)	S	T (s)	S	T (s)	S
interdom	79.91	2.92x	61.34	16.97x	57.23	17.28x	OOT	★	OOT	★	OOT	★
amazon0505	3.57	15.90x	0.09	3.81x	0.01	1.00x	0.347	11.98x	*0.063	0.21x	*0.05	0.17x
email-euAll	8.72	18.05x	1.84	17.14x	0.10	6.33x	164.47	18.88x	4.29	17.9x	*0.04	0.16x
epinions1	165.82	17.76x	90.57	17.06x	15.72	18.07x	18496.36	★	2869.65	★	186.02	19.3x
slashdot090221	55.63	17.80x	24.06	17.00x	15.61	16.68x	2800.90	★	1513.92	16.46x	165.65	18.7x
wiki-vote	28.68	18.55x	14.04	18.72x	0.91	18.37x	2224.84	★	140.79	19.63x	0.01	4.3x

Table 4: Parallel execution time and speedup with respect to sequential version. ★: sequential algorithm ran out of time.

values of speedup for all the situations in which the sequential running time is not negligible (i.e. more that 1 seconds). Reasonably, if this is not the case, the overhead of the parallelization reduces (or eliminates) the benefits of the parallel execution.

In general, the parallel implementation gives an almost linear (in the number of used threads) reduction of the execution time. This allowed us to complete the execution within the deadline of a higher number of scenarios with respect to the case of the sequential implementation (i.e. all the 2-plexes cases and all the 3-plexes with the exception of `interdom`). Achieving such a good speedup has been possible thanks to a careful design of the sequential algorithm. For example, due to the efficient pruning performed on the BLOCKS, the size of the data to keep in the processors' caches is reduced, decreasing contention effects on the last level caches and improving the scalability of the parallel implementation.

Furthermore, the parallel implementation enables the enumeration of meaningful k-plexes on the biggest graphs of Table 1. Table 5 shows the obtained execution times.

Concerning the support for distributed memory clusters of multicore machines, we show in Figure 5 the speedup we achieve for different numbers of machines with respect to a single computing machine. In this case the speedup is defined as the execution time when using one machine (with 20 cores), divided by the execution time when using a given number of machines (each one with 20 cores). For this test we selected some of the biggest graphs, since it would not be very meaningful to use a cluster for problems which takes few seconds to be computed on a single parallel machine. As depicted by the Figure, the speedup differs according to the graph.

Network	q	Time (s)	
		2-plexes	3-plexes
as-skitter	50	3531.87	OOT
	100	0.24	0.25
in-2004	50	43.51	OOT
	100	18.209	9720.47
pokec	30	0.63	0.82
	50	*0.48	*0.6
uk-2005	250	1567.26	OOT
	500	16.14	190.6
it-2004	2000	125.53	OOT
	3000	18.59	407.49

Table 5: Parallel execution time over big graphs using 20 threads. *: there were no solutions.

In some cases (e.g. `wiki-vote`) by using 16 multicore machines we achieve a speedup of ~12 with respect to the single machine. Considering that on the single machine we had a speedup of ~19 with respect to the sequential version, we reduced the execution time by ~230 times with respect to the sequential implementation.

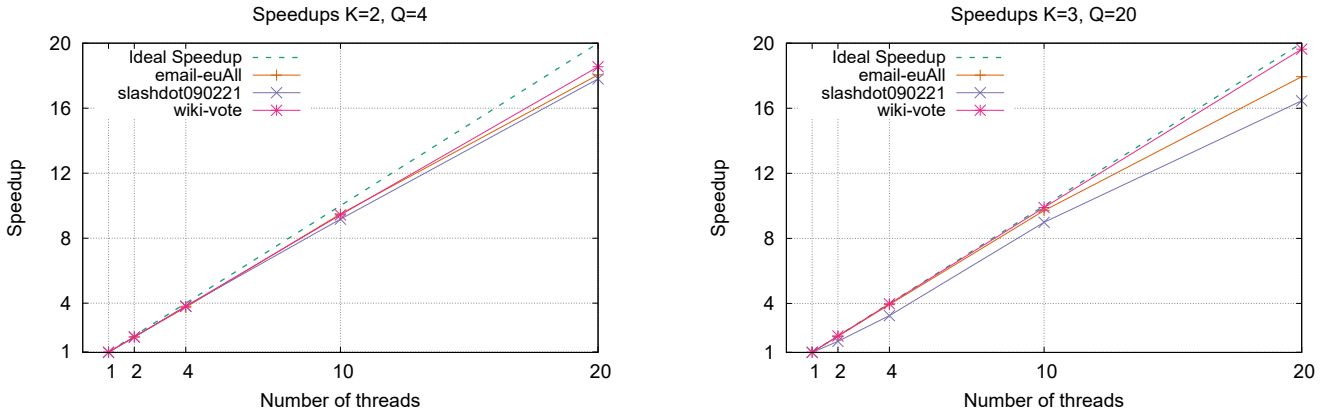

Figure 4: Speedup of the parallel over the sequential version

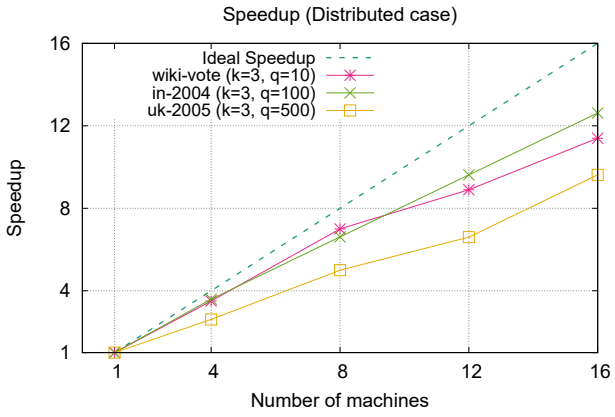

Figure 5: Speedup of the distributed case

An interesting study case involves the computation of 3-plexes with $q = 50$ for `in-2004`. While a parallel implementation on a single machine was not able to terminate in 12 hours, our distributed implementation using 16 machines was able to find all the k-plexes in less than 3 hours. As the experiments suggest a close to ideal scaling factor, we may conjecture that a sequential implementation of the algorithm would have required several weeks to find the same result.

6 RELATED WORK

The papers by Bron and Kerbosch [6] and Tsukiyama et al. [22] are at the heart of many algorithms for enumerating cliques and their relaxed versions such as k-plexes due to their effectiveness. The Bron-Kerbosch algorithm relies on a backtracking scheme that is adopted in several efficient algorithms due to its simplicity and good practical performance [12, 20, 26]. While the original version [6] does not provide any guarantee, the version in [20] guarantees a total running time of $O(3^{n/3})$, which worst-case-optimal, and the one in [12] further improves the work for sparse graphs, which may have up to $(n-d)3^{d/3}$ maximal cliques, by producing an algorithm with $O(d(n-d)3^{d/3})$ time. This strategy has been adapted to the enumeration of maximal k-plexes in [25], and inspired the similar backtracking structure in [24].

The algorithm by Tsukiyama et al. has been originally conceived for the equivalent problem of enumerating maximal independent sets, and has been subsequently adapted to maximal clique enumeration by Chiba-Nishizeki [8]. Makino-Uno [16] has reinterpreted some of its ideas in the paradigm of reverse search introduced by Avis and Fukuda [1]. A space efficient algorithm with bounded delay has been given in [10]. A simplified version of this strategy has been applied to the enumeration of maximal k-plexes in [3], which proposed the first output-sensitive (i.e., whose time is proportional to the number of solutions found) k-plex enumeration algorithm, and then further exploited by [9] for large k-plexes.

As k-plexes generalize cliques, their discovery takes more time than finding cliques. For this reason several algorithms for listing *all* k-plexes [3, 25] do not scale well with large graphs. For example, [3] needs to maintain a large database of all maximal k-plexes found so far to avoid duplicates, which can create contention and excessive space usage. A new trend has emerged in recent years to simultaneously speed up the computation and avoid finding solutions of little interest for the final user: that is looking for a reduced number of just *significant* results. For cliques, an example is the work by Zhou et al. for finding the top-k cliques [28]. Zhai et al. [27], rather than aiming for large sized solutions, define a structure similar to the k-plex, where small solutions are allowed as long as they are dense enough. Liu and Wong [15] also propose a variant of the k-plex, in which k varies with the size of the solution, which allows for search space cuts based on node degree. Behar and Cohen [2] aim at finding large connected s-cliques, i.e., connected sets of nodes within distance s of each other. As for k-plexes, examples are [24] and [9]. Both algorithms include a *minimum size* parameter which cuts of small solutions from the search, with hard-coded lower bounds in their structure to improve their efficiency: [9] finds k-plexes with at least k^2 nodes, and after a careful reading of the paper (and as confirmed by the authors in private communications) it can be seen that [24] finds k-plexes with at least $k+1$ nodes. It should be noted that we lack good theory to understand why these algorithms can work efficiently on real-world data as some of them could require exponential time in the worst case.

Finally, it should be remarked that subgraph enumeration is one of several techniques for extracting community structure from networks. Other examples include k-core [18] and k-truss decomposition [14], which aim at removing sparse parts of the networks, leaving only the most interconnected areas. A comprehensive survey of the area can be found in [13].

7 CONCLUSION

We proposed D2K, the first algorithm that can find k-plexes of very large real world networks, by an effective combination of algorithm design and insight on the problem. To maximize this benefit, we proposed a parallel and distributed implementation which scales up nicely to tens of machines, with tens of cores each, further extending the reach of D2K to networks whose sequential processing time would otherwise be massive. This allowed to compute for the first time k-plexes in large real-world networks.

D2K moves an important chunk of today's real data within the reach of pseudo-clique detection, improving the applicability of existing network analysis methods. As a toy example, we provide some insight on the largest communities in two networks with half a billion edges, and further validate our algorithm on real networks of different types. We believe that D2K, with its open source implementation, sets a milestone for dense subgraph enumeration, and will open new directions for community discovery in large graphs.

ACKNOWLEDGEMENTS

We wish to thank the authors of [3, 9, 24] for sharing their source code, prof. Marco Danelutto and the people at *ITC Center* of the University of Pisa for providing the machines used in the experiments, and the reviewers for giving some useful comments. The work was partially supported by JST CREST, Grant Number JPMJCR1401, Japan and by EU H2020-ICT-2014-1 project RePhrase (No. 644235).

REFERENCES

[1] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21 – 46, 1996.
[2] Rachel Behar and Sara Cohen. Finding all maximal connected s-cliques in social networks. In *21th International Conference on Extending Database Technology*, EDBT, pages 61–72, 2018.
[3] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. Efficient enumeration of maximal k-plexes. In *2015 ACM SIGMOD International Conference on Management of Data*, pages 431–444. ACM, 2015.
[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.
[5] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.
[6] Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Communications of the ACM*, 16(9):575–576, 1973.
[7] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. Fast algorithms for maximal clique enumeration with limited memory. In *18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, pages 1240–1248, 2012.
[8] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.
[9] Alessio Conte, Donatella Firmani, Caterina Mordente, Maurizio Patrignani, and Riccardo Torlone. Fast enumeration of large k-plexes. In *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, pages 115–124, 2017.
[10] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming*, ICALP, pages 148:1–148:15, 2016.
[11] Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, and Riccardo Torlone. Finding all maximal cliques in very large social networks. In *19th International Conference on Extending Database Technology*, EDBT, pages 173–184, 2016.
[12] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18, 2013.
[13] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75 – 174, 2010.
[14] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *2014 ACM SIGMOD International Conference on Management of Data*, pages 1311–1322. ACM, 2014.
[15] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. In *Machine Learning and Knowledge Discovery in Databases*, pages 33–49. Springer Berlin Heidelberg, 2008.
[16] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Algorithm Theory-SWAT 2004*, pages 260–272. Springer, 2004.
[17] Robert J Mokken. Cliques, clubs and clans. *Quality and quantity*, 13(2):161–173, 1979.
[18] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2013.
[19] Stephen B Seidman and Brian L Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, 6(1):139–154, 1978.
[20] 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.
[21] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013*, pages 104–112, 2013.
[22] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.
[23] Takeaki Uno. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica*, 56(1):3–16, 2008.
[24] Zhuo Wang, Qun Chen, Boyi Hou, Bo Suo, Zhanhuai Li, Wei Pan, and Zachary G. Ives. Parallelizing maximal clique and k-plex enumeration over graph data. *Journal of Parallel and Distributed Computing*, 106:79 – 91, 2017.
[25] Bin Wu and Xin Pei. A parallel algorithm for enumerating all the maximal k-plexes. In *Emerging Technologies in Knowledge Discovery and Data Mining*, pages 476–483, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
[26] Yanyan Xu, James Cheng, Ada Wai-Chee Fu, and Yingyi Bu. Distributed maximal clique computation. In *2014 IEEE International Congress on Big Data*, pages 160–167. IEEE, 2014.
[27] Hongjie Zhai, Makoto Haraguchi, Yoshiaki Okubo, and Etsuji Tomita. A fast and complete algorithm for enumerating pseudo-cliques in large graphs. *International Journal of Data Science and Analytics*, 2(3):145–158, Dec 2016.
[28] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. Finding top-k maximal cliques in an uncertain graph. In *26th International Conference on Data Engineering*, ICDE, pages 649–652. IEEE, 2010.