Discovering Dense Subgraphs in Parallel for Compressing Web and Social Networks[⋆,⋆⋆]

Cecilia Hernández[1,2] and Mauricio Marín[3]

[1] Dept. of Computer Science, University of Concepción, Chile
[2] Dept. of Computer Science, University of Chile, Chile
[3] Yahoo Research, Santiago
chernand@dcc.uchile, mmarin@yahoo.com

Abstract. Mining and analyzing graphs are challenging tasks, especially with today's fast-growing graphs such as Web and social networks. In the case of Web and social networks an effective approach have been using compressed representations that enable basic navigation over the compressed structure. In this paper, we first present a parallel algorithm for reducing the number of edges of Web graphs adding virtual nodes over a cluster using BSP (Bulk Synchronous Processing) model. Applying another compression technique on edge-reduced Web graphs we achieve the best state-of-the-art space/time tradeoff for accessing out/in-neighbors. Second, we present a scalable parallel algorithm over BSP for extracting dense subgraphs and represent them with compact data structures. Our algorithm uses summarized information for implementing dynamic load balance avoiding idle time on processors. We show that our algorithms are scalable and keep compression efficiency.

Keywords: Parallel algorithms, Compressed Web and social graphs.

1 Introduction

Massive graphs appear in a wide range of domains including the Web, social networks, RDF graphs, protein networks and many more. For instance, the Web graph on a recent estimation has more than 7.8 billion pages with more than 200 billions of edges (mentioned in previous work [1]).

In the last decade, many graph algorithms have been proposed to address some of the problems associated with large graphs. Different approaches have been used to manage large graphs. One approach consists of representing graphs in compressed form while being able to resolve queries of interest without decompression. Although these compressed structures are usually slower than uncompressed representations, they are faster than having to access the disk. Many of these compressed structures target Web graphs, some support out-neighbor queries [2,3], that is retrieving the outgoing links of a node x, and some also

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-319-02432-5_33

[⋆] Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F.

[⋆⋆] Partially funded by FONDEF IDeA CA12i10314.

support in-neighbor queries, that is incoming links of a node x [4]. Another approach is the use of distributed systems where distributed memory is aggregated to process the graph. Distributed memory is useful when data is larger than the memory available on a commodity machine. Pregel [5] is a graph system that works on BSP model, Pegasus [6] is a graph mining library over Hadoop, which is the free implementation of MapReduce [7]. Pace [8] discusses important differences between BSP and MapReduce and shows that iterative algorithms are more efficient using BSP than MapReduce.

The main contributions of this paper are:

– A scalable BSP parallel algorithm for reducing the number of edges of Web graphs by finding dense subgraphs and adding virtual nodes. This algorithm is based on DSM (Dense Subgraph Mining) algorithm, which used with virtual nodes, BFS ordering and K2tree [9] achieves the best compression on Web graphs [1].
– A scalable parallel DSM algorithm for extracting dense subgraphs. The algorithm exploits locality of adjacency lists and uses dynamic load balanced for maximizing processor utilization avoiding idle times. Representing these dense subgraphs with compact data structures [10] combined with an improved version of MPk [11] provides the best space/time tradeoffs for social networks [10].

2 Related Work

Compressing the Web has been an active research area for some time. Some of the earlier proposals include basic navigation, which is reduced at retrieving out-neighbors [2,3], and others that include retrieving out/in-neighbors [4,11,10]. Compression techniques for Web graphs use different patterns, such as locality and similarity of adjacency lists [2], the sparse nature of the adjacency matrix [4], label ordering [3,2], edge reduction [12], and dense subgraphs [10,1]. In social networks, successful representations use clique-like structures [13,11] and more dense subgraph patterns, such as cliques, bicliques and other patterns that combine cliques and bicliques [10]. Some of these structures [11,10,1] use compact data structures based on bit vectors and symbol sequences. Compact data structures use space efficiently and their basic operations are rank/select/access.

Discovering dense subgraphs in large graphs is a challenging problem in data analysis and has a wide-range of applications, including community mining, spam detection, and social analysis. The general problem has many variants such as finding and enumerating cliques [14] and detecting dense subgraphs or communities [15,16]. Although, there are some differences in the terminology defining a dense subgraph, all works consider the density as measuring the number of edges in relation with the number of nodes in such structures.

In recent years, parallel and distributed data management has gained attention due to the success of MapReduce [7] and Hadoop. MapReduce is simple to use and provides high throughput. Pregel [5] aims processing graphs and it is based on vertex computation using BSP. However, MapReduce and Pregel require hundreds or thousands of machines in order to process large graphs. For instance, Pegasus [6] and Pregel focus on large graph querying and mining, Pegasus is built on top of Hadoop and Pregel is built using BSP. Pregel improves upon MapReduce by passing computation results instead of graph structures among processors.

3 Our Approach

We represent a web graph as a directed graph $G = (V, E)$ where V is a set of vertices (pages) and $E \subseteq V \times V$ is a set of edges (hyperlinks). For an edge *e=(u,v)*, we call u the *source* and v the *center* of e. We find patterns given by the following definition.

Definition 1. A *dense subgraph* $H(S, C)$ of $G = (V, E)$ is a graph $G'(S \cup C, S \times C)$, where $S, C \subseteq V$.

Note that this definition includes cliques ($S = C$) and bicliques ($S \cap C = \emptyset$). Our goal is to represent the $|S| \cdot |C|$ edges of a dense subgraph $H(S, C)$ in space proportional to $|S| + |C| - |S \cap C|$. Thus, the bigger the dense subgraphs we detect, the more space we save at representing their edges.

The parallel algorithms presented here are based on a sequential algorithm for discovering dense subgraphs, DSM (Dense Subgraph Mining) [1]. DSM consists of 2-step clustering and 2-step mining. The clustering algorithm computes $|R|$ hash values for each adjacency list conforming a matrix of hash values of dimension $|R \cdot V|$ (Step 1). The matrix is sorted by columns where each cluster is formed by similar rows (Step 2). The mining phase takes the adjacency lists related to hash rows of each cluster and sorts edges by frequency (Step 3). Then, each adjacency list of the cluster is inserted into a prefix tree, discarding edges of frequency 1. Each node v in the prefix tree has a label (consisting of the node id), and it represents the sequence $l(v)$ of labels from the root to the node. Such node v stores also the range of graph nodes whose list start with $l(v)$ (Step 4). Figure 1 shows an example.

Our first parallel algorithm (Algorithm 1 in Table 1) uses DSM for reducing edges by a factor between 5 and 10, adding a small percentage of virtual nodes

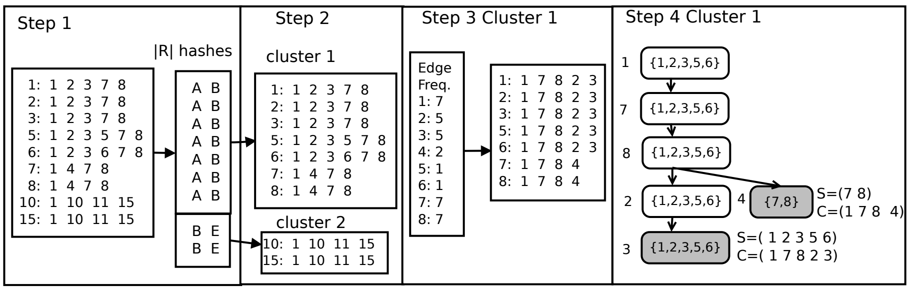

Fig. 1. Example of the Dense Subgraph Mining (DSM) algorithm

(around 10 and 15 %). Then, we apply BFS ordering and k2tree over the edge-reduced Web graphs. Our second parallel algorithm (Algorithm 2 in Table 1) uses DSM for extracting dense subgraphs and represent them with compact data structures. Such representation is based on the following components.

Let $\mathcal{H} = \{H_1, \ldots, H_N\}$ be the dense subgraph collection found in the graph, based on Definition 1. We represent $\mathcal{H}$ as a sequence of integers X with a corresponding bitmap B. Sequence $X = X_1 : X_2 : \ldots : X_N$ represents the sequence of dense subgraphs and bitmap $B = B_1 : B_2 : \ldots B_N$ is used to mark the separation between each subgraph. We now describe how a given X_r and B_r represent the dense subgraph $H_r = H(S_r, C_r)$.

We define X_r and B_r based on the overlapping between the sets S and C. Sequence X_r will have three components: L, M, and R, written one after the other in this order. Component L lists the elements of $S - C$. Component M lists the elements of $S \cap C$. Finally, component R lists the elements of $C - S$. Bitmap $B_r = 10^{|L|}10^{|M|}10^{|R|}$ gives alignment information to determine the limits of the components. In this way, we avoid repeating nodes in the intersection, and have sufficient information to determine all the edges of the dense subgraph.

Table 1. Algorithms DSM with virtual nodes (Algorithm 1), and DSM for extracting dense subgraphs (Algorithm 2)

```
Algorithm 1
Input: $G_p$, $ES$, $T$
Output: Reduced $RG(|V + VN|, E2)$ graph
  Each processor reads its data partition
  {Step 0}
  for ($i \leftarrow 0$ to $T - 1$ ) do
    $clusters = FindClusters()$
    for ($c \in clusters$) do
      $Sets(S, C) = FindDenseSubs(c, ES)$
      $localVnodes = DefineSets(S, C)$
      $Replace(G_p, Sets(S, C), localVnodes)$
      $AddVnodes(G_p, Sets(S, C), localVnodes)$
    end for
  end for
  $sendLocalVnodeMsg()$
  $sync()$
  {Step 1}
  if ($proc == 0$) then
    $lvnodes = RecibeMsgs()$
    $gvnodes = ProcVNodeGlobal(lvnodes)$
    $sendGlobalVnodes(gvnodes)$
  end if
  $sync()$
  {Step 2}
  $gvnodes = RecieveMsgs()$
  $replaceVnodes(G_p, lvnodes, gvnodes)$
  return $RG$
```

```
Algorithm 2
Input: $G_p$, $esArray$, $T$, $threshold$.
Output: Dense subgraph collection
  Each processor reads its data partition
  $ES = esArray.first()$
  {Step 0}
  for ($i \leftarrow 0$ to $T$) do
    $clusters = FindClusters()$
    for ($c \in clusters$) do
      $Sets(S, C) = FindDenseSubs(c, ES)$
      $numDSs = |Sets(S, C)|$
      $WriteToDisk(Sets(S, C))$
    end for
    if ($i == period()$) then
      $sendLoadMsg()$
      $sync()$
      {Step 1}
      if ($proc == 0$) then
        $ProcessLoad()$
        $sendDistInfo()$ (to all procs)
      end if
      $sync()$
    end if
    {Step 2}
    $sendData()$
    if ($numDSs < threshold$) then
      $ES = esArray.next$
    end if
  end for
```

3.1 Algorithms and Analysis

The BSP model provides an efficient parallel distributed memory model that considers relevant parameters of a real parallel computer system. A BSP computer is defined by P processors with local memory, connected via a point-to-point communication link. BSP algorithms proceed in supersteps in each of which processors receive input data, perform asynchronous computation over its data and communicate output at the end. Supersteps are synchronized at the end using barriers. An algorithm designed in BSP is measured by three main features: *computation*, *communication*, and *synchronization* costs. The cost model is given by $W + Hg + L$, where W is the maximum cost of computation on a processor, H is the maximum input/output communicated among processors, g is the latency and L is the synchronization cost.

Algorithm 1 in Table 1-(left) describes our parallel DSM for reducing edges and adding virtual nodes. During **Step 0** each processor processes G_p in parallel locally. Each iteration finds all clusters on G_p and on each cluster the mining algorithm discovers dense subgraphs of the type $\mathcal{H}$ with components (S,C) of size at least ES. For each subgraph, we create local virtual node ids (*localVnodes*) to separate sets (S,C).

In **Step 0** all processors sends a tuple with $(lvnodeInit, numberLVnodes)$ to processor 0. Thus, **Step 0** is $O(O(T\frac{|E|}{P}\log\frac{|E|}{P})$, where $|E|$ is the number of edges in G, P the number of available processors, and T the number of iterations. In **Step 1** processor 0 relabels local virtual nodes to global ids and sends that information to all processors. Relabeling is done by changing the $gVnodeInit$ based on the number of virtual nodes found in previous processed processor tuple, that is $gVnodeInit_i = vninit$ and $gVnodeInit_{i+1} = \sum numberLVnodes_i$. We work with global virtual node ids instead of locals to minimize mapping space. Thus, **Step 1** is $O(Pg + L)$. In **Step 2** all processors receive tuples with global virtual node ids and each processor replaces local virtual node ids for global ones. Then, this step is $O(|V + VN|)$ which indicates that the algorithm scales up efficiently since the amount of required communication is much smaller than the amount of computation performed by processors on local data. Therefore, the total cost is $O(T(\frac{|E|}{P}\log\frac{|E|}{P}) + Pg + L + |V + VN|)$.

Algorithm 2 in Table 1-(right) describes our parallel algorithm for extracting dense subgraphs using dynamic load balance. This is an iterative algorithm, where each iteration has several steps. In **Step 0** each processor computes clustering and mining and extracts dense subgraphs and sends periodically its workload information to processor 0. Processor workload tuple is given by ES and $numDSs$, where ES is the current size of the dense subgraphs that are mined and $numDSs$ is the number of subgraphs at the current iteration. The clustering is $O(\frac{|E|}{P}\log\frac{|E|}{P})$ and all processors send local workload tuples to processor 0 in $O(Pg + L)$ periodically. Function *period*() determines how often processors send their load. In **Step 1** processor 0 receives local load from all processors, computes a global load tuple containing $(minP, maxP, minES, maxES, minDSs, maxDSs)$, and decides whether load balance is performed and the amount of data to move. If it decides to apply load balance, it sends global load balance tuple to all processors. In **Step 2** each processor receives the global load tuple and the heavier processor sends a portion of its data to the lighter processor.

Step 0 is computed T times and each processor sends workload tuples to processor 0 T_p times. During *Step 1* processor 0 computes workload tuples and decide whether heavier processors will send data to lighter processors, which is $O(Pg + L)$. Applying load balance depends on the distance between $(maxES, minES)$ and $(minDSs, maxDSs)$ among processors, and it can happen T_d times. This step is computed in $O(Mg + L)$, where M is a portion of G_p to move. The total cost is $O(T(\frac{|E|}{P} \log \frac{|E|}{P}) + T_p(Pg + L + P) + T_d((P + M)g + L))$.

4 Experimental Evaluation

We perform different experiments over Web and social graphs described in Table 2. [1]. We use the natural order for input graphs in all our experiments. We implemented parallel algorithms using C++ and BSP over a cluster with at most 64 processors. Each processor is an Intel 2.66 GHz, with 24 GB of RAM and 8 MB of cache. We partition input graphs among processors by equal number of edges contained by complete list of out-neighbors. This partition scheme gave us more balanced processor work load.

We study the performance of our parallel DSM with virtual nodes and extracting dense subgraphs using dynamic load balance. We analyze the effect of using different number of processors in terms of compression efficiency, running times, and speedup. We also compute the Edge ratio (ER). ER is the total number of edges (belonging to dense subgraphs) extracted in parallel versus the total number of edges in dense subgraphs extracted with the sequential algorithm.

Table 2. Number nodes, edges and size in MBs of graphs. A1S stands for the speedup(S) for 8 and 64 processors when using DSM in Algorithm 1, and A2S when using DSM in Algorithm 2. ER (Edge ratio) is the number of edges (belonging to dense subgraphs) extracted in parallel versus the ones extracted sequentially.

Data Set	Nodes	Edges	MB	A1S (8)	A1S (64)	A2S (8)	ER	A2S (64)	ER
eu-2005	862,664	19,235,140	77	4.95	20.88	14.46	0.99	58.6	0.99
indochina	7,414,866	194,109,311	765	2.18	23.85	5.39	0.96	52.4	0.99
uk-2002	18,520,486	298,113,762	1,200	10.10	68.18	11.21	0.90	102.87	0.97
arabic-2005	22,744,080	639,999,458	2,500	10.52	55.40	6.95	0.92	66.80	0.96
dblp-2011	986,324	6,707,236	30	-	-	29.08	0.76	117	0.87
LJSNAP	4,847,571	68,993,773	280	-	-	8.63	0.45	55.34	0.65

Figure 2 shows parallel running times and compression performance (bpe) using different numbers of processors for different Web graphs. We include running times for computing DSM-ESx-T10 (where $ES = x$ for finding dense subgraphs of at least size x, and $T = 10$ i.e. 10 iterations); and the running time for achieving the complete compression structure, which consists of two parts; DSM-ESx-T10 builds a graph with fewer edges and virtual nodes (RG); and K2treeBFS applies BFS and k2tree over RG. As observed, the running time improves greatly without affecting compression. These results suggest that there

[1] Data sets available at: `law.dsi.unimi.it` and `snap.stanford.edu/data`

is a great amount of locality of reference in adjacency lists. Figure 2 shows that the cost of applying k2treeBFS, which is sequential, has more impact on larger graphs. This is seen by the distance between the two running time plots visible on Arabic data set.

Table 2 shows the speedup achieved using 8 and 64 processors (A1S (8) and A1S (64)) using DSM with virtual nodes (k2tree not included). We observe that the speedup is higher for larger graphs, which suggest that such graphs take more advantage of memory aggregation in the cluster system.

We evaluate our second parallel algorithm (extracting dense subgraphs with DSM) measuring running times, speedup and the Edge ratio (ER). We use 100 iterations for extracting dense subgraphs and dblp-2011 and 200 iterations for LJSNAP (LiveJournal). Figure 3 shows the running time for DSM with dense subgraph extraction, considering only time for extraction, complete compression time (including mpk), and the compression achieved for social networks using $\mathcal{H}$ and $\mathcal{R}$ with *mpk* [11]. This figure also shows that the sequential part of the compression construction slows down the compression time. Table 2 shows the speedup for 8 and 64 processors (A2S (8) and A2S (64)). We also measure ER, which is the total number of edges extracted in dense subgraphs using our parallel algorithm versus the total number of edges extracted belonging to dense subgraphs using the sequential algorithm. We extract in parallel more than 90% edges belonging to dense subgraphs achieving good speedups on Web graphs. However, it is less effective on social graphs where ER is lower as seen in Table2.

5 Conclusions

This paper proposes two parallel algorithms for DSM, a sequential algorithm for discovering dense subgraphs [1] for compressing Web and social graphs. Our first parallel algorithm uses DSM with virtual nodes for reducing the number of edges. This algorithm exploits locality of reference of adjacency lists. Applying BFS ordering and k2tree over parallel edge-reduced Web graphs does not degrade compression efficiency. Our second parallel algorithm extracts dense subgraphs

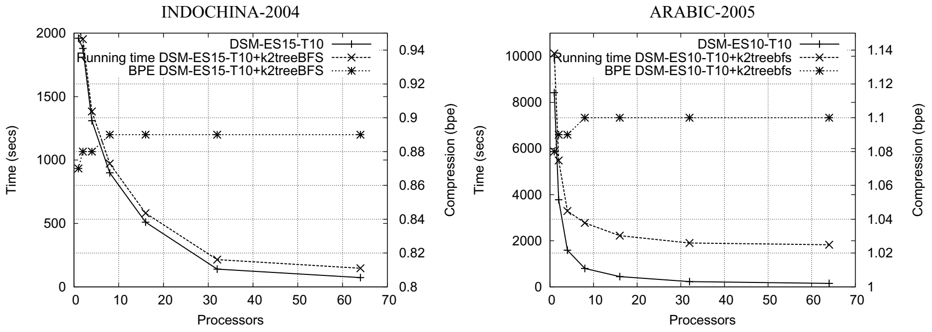

Fig. 2. Parallel running time with corresponding compression for Web graphs

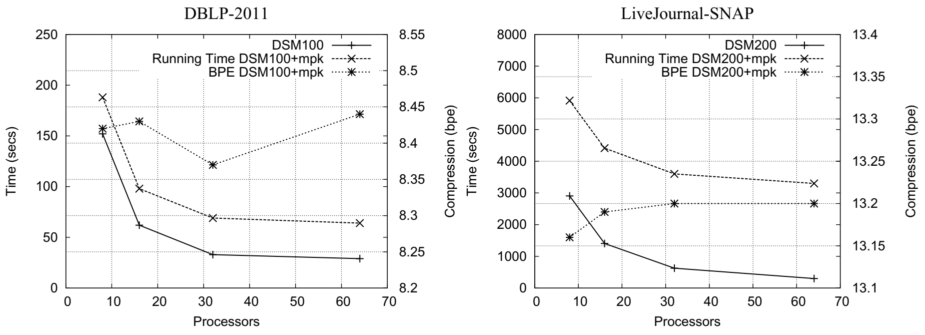

Fig. 3. Parallel running time for extracting dense subgraphs and bpe for social graphs

in parallel using dynamic load balance. Both algorithms provide good speedup and compression efficiency. However, since both algorithms are used with other sequential compression techniques such as *k2tree* [9] and *mpk* [11], they limit our compression speed.

References

1. Hernández, C., Navarro, G.: Compressed representations for web and social graphs. To appear in Knowledge and Information Systems (2013), `http://link.springer.com/article/10.1007/s10115-013-0648-4`
2. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: WWW, pp. 587–596 (2011)
3. Apostolico, A., Drovandi, G.: Graph compression by bfs. Algorithms 2(3), 1031–1044 (2009)
4. Brisaboa, N.R., Ladra, S., Navarro, G.: k^2-Trees for compact web graph representation. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 18–30. Springer, Heidelberg (2009)
5. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: SIGMOD Conference, pp. 135–146 (2010)
6. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: mining peta-scale graphs. Knowl. Inf. Syst. 27(2), 303–325 (2011)
7. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
8. Pace, M.F.: Bsp vs mapreduce. Procedia CS 9, 246–255 (2012)
9. Ladra, S.: Algorithms and compressed data structures for information retrieval. Ph.D. Thesis, University of A. Coruña (2011)
10. Hernández, C., Navarro, G.: Compressed representation of web and social networks via dense subgraphs. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 264–276. Springer, Heidelberg (2012)
11. Claude, F., Ladra, S.: Practical representations for web and social graphs. In: CIKM, pp. 1185–1190 (2011)

12. Buehrer, G., Chellapilla, K.: A scalable pattern mining approach to web graph compression with communities. In: WSDM, pp. 95–106 (2008)
13. Maserrat, H., Pei, J.: Neighbor query friendly compression of social networks. In: KDD, pp. 533–542 (2010)
14. Schmidt, M.C., Samatova, N.F., Thomas, K., Park, B.-H.: A scalable, parallel algorithm for maximal clique enumeration. J. Parallel Distrib. Comput. 69(4), 417–428 (2009)
15. Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: Trawling the Web for emerging cyber-communities. Computer Networks 31(11-16), 1481–1493 (1999)
16. Dourisboure, Y., Geraci, F., Pellegrini, M.: Extraction and classification of dense communities in the web. In: WWW, pp. 461–470 (2007)