A GPU-accelerated local search algorithm for the Correlation Clustering problem

Mario Levorato
Department of Computer Science, Fluminense Federal University
24210-240 Niterói, RJ – Brasil
mlevorato@ic.uff.br

Lúcia Drummond and Yuri Frota
Department of Computer Science, Fluminense Federal University
24210-240 Niterói, RJ – Brasil
{lucia, yuri}@ic.uff.br

Rosa Figueiredo
Laboratoire d'Informatique d'Avignon, University of Avignon
84911 Avignon, France
rosa.figueiredo@univ-avignon.fr

RESUMO

A solução ótima para o problema de Correlação de Clusters (*Correlation Clustering* ou CC) pode ser utilizada como medida do nível de equilíbrio em redes sociais de sinais, onde interações positivas (amizade) e negativas (antagonismo) estão presentes. Metaheurísticas têm sido utilizadas com sucesso para resolver não apenas este, como também outros problemas difícies de otimização combinatória, por serem capazes de fornecer soluções sub-ótimas em um tempo razoável. Este trabalho propõe uma implementação alternativa de busca local baseada em GPGPUs, a qual pode ser utilizada em conjunto com as metaheurísticas GRASP e ILS existentes para o problema CC. Esta nova abordagem supera, em tempo de execução, o procedimento de busca local até então aplicado, com qualidade de solução similar, apresentando speedups médios que vão de x1.8 a x28.

PALAVRAS CHAVE. CUDA, GPGPU, VND, GRASP, ILS, Correlação de Clusters.

Área Principal: MH - Metaheurísticas

ABSTRACT

The solution of the Correlation Clustering (CC) problem can be used as a criterion to measure the amount of balance in signed social networks, where positive (friendly) and negative (antagonistic) interactions take place. Metaheuristics have been used successfully for solving not only this problem, as well as other hard combinatorial optimization problems, since they can provide sub-optimal solutions in a reasonable time. In this work, we present an alternative local search implementation based on GPGPUs, which can be used with existing GRASP and ILS metaheuristics for the CC problem. This new approach outperforms the existing local search procedure in execution time, with similar solution quality, presenting average speedups from x1.8 to x28.

KEYWORDS. CUDA, GPGPU, VND, GRASP, ILS, Correlation Clustering.

Main Area: MH - Metaheuristics

1. Introduction

Structural (or social) balance is considered a fundamental social process. It has been used to explain how the feelings, attitudes and beliefs, which the social actors have towards each other, can promote the formation of stable (but not necessarily conflict-free) social groups. The balance of a social system tends to follow the human tendency to preserve a cognitive consistency of hostility and friendship. The principle is simple: "my friend's friend is my friend, my friend's enemy is my enemy, my enemy's friend is my enemy, my enemy's enemy is my friend" (Heider, 1946). Absence of balance creates a kind of tension in the group members' minds that can eventually lead to changes in their opinions. Once balance is achieved, it tends to be stable, since no cognitive dissonance could change the state (Hummon and Doreian, 2003).

Determining the structural balance of a signed social network has been a key aspect in the study of the structure and origin of tensions and conflicts in a network of individuals whose mutual relationships are characterizable in terms of friendship and hostility. Structural balance theory was first formulated by Heider (1946) with the purpose of describing sentiment relations between people pertaining to a same social group (like/dislike, love/hate, trust/distrust). Signed graphs were then introduced by Cartwright and Harary (1956), who formalized Heider's theory stating that a balanced social group could be partitioned into two mutually hostile subgroups each having internal solidarity. In the last decades, signed graphs have shown to be a very attractive discrete structure for social network researchers (Doreian and Mrvar, 1996; Inohara, 1998; Yang et al., 2007; Abell and Ludwig, 2009). Different criteria and solution approaches have been used in the literature so as to quantify and evaluate balance in a signed social network (Doreian and Mrvar, 2009; Leskovec et al., 2010; Facchetti et al., 2011; Srinivasan, 2011).

Clustering is the action of partitioning individual elements into groups based on their similarity. Clustering problems defined on signed graphs arise in many scientific areas (Bansal et al., 2002; Gülpinar et al., 2004; DasGupta et al., 2007; Traag and Bruggeman, 2009; Huffner et al., 2010; Macon et al., 2012; Figueiredo and Frota, 2014). The common element among these applications is the collaborative *vs.* conflicting environment in which they are defined. The solution of clustering problems defined on signed graphs can be used as a criteria to measure the degree of balance in social networks (Doreian and Mrvar, 1996, 2009; Figueiredo and Moura, 2013). By considering the original definition (Heider, 1946) of structural balance, the optimal solution of the very known Correlation Clustering (CC) Problem (Bansal et al., 2002) arises as a measure for the degree of balance in a social network. Other applications of the CC Problem include efficient document classification (Bansal et al., 2002), detection of embedded matrix structures (Gülpinar et al., 2004), biological systems (DasGupta et al., 2007), portfolio analysis in risk management (Huffner et al., 2010) and image segmentation (Kim et al., 2014).

From a practical point of view, in solving the clustering problem treated in this paper, heuristic approaches are primarily of interest, since large social networks may have to be analyzed (Kunegis et al., 2009; Leskovec et al., 2010; Facchetti et al., 2011). For example, online networks with two opposite kinds of relationships are nowadays very common. Slashdot, a technology-related news website, includes a feature which allows users to tag each other as friends or foes, thus allowing users to rate other users negatively. On online review websites such as Epinions users can either like or dislike other people's reviews. This behavior can be modeled as a signed network, where edge weights can be either greater or less than 0, representing positive or negative relationships respectively. The definition of a measure to represent the imbalance of a social network adds itself a degree of approximation to the task of evaluating balance in a social network. Thus, it is imperative that the clustering problem associated with this measure be solved efficiently.

To our knowledge, there are three metaheuristic approaches applied to the CC problem. Zhang et al. (2008) proposes genetic algorithms to the CC problem, with an application to document clustering. This strategy was impossible to reproduce though, for the absence of information about how the genetic operators are applied. Drummond et al. (2013) presents a Greedy Randomized

Adaptive Search Procedure (GRASP) (Feo and Resende, 1995) to evaluate structural balance in signed social networks. Later, based on this work, Levorato et al. (2014) introduced an Iterated Local Search (ILS) (Lourenço et al., 2003) metaheuristic for the CC problem, which outperformed, in processing time, the GRASP metaheuristic proposed earlier, with similar or improved solution quality. By observing the great amount of time spent on the processing of larger graphs, we saw an opportunity to extend the aforementioned GRASP and ILS algorithms with a new implementation of local search that can solve the problem faster.

In this work, we present a parallel local search procedure for the CC problem, accelerated by General Purpose Graphics Processing Units (GPGPUs). Then, by applying the proposed local seach in the GRASP and ILS metaheuristics, we show the improvements over the existing sequential local search procedure. We believe that the lessons learned with this local search parallelization on the GPU can be used in other related problems. The paper is organized as follows. Section 2 presents the Correlation Clustering problem, including a mathematical formulation and a literature review of it. Section 3 describes the parallel local search algorithm for the CC problem that runs on the GPU, while Section 4 lists the experimental results of it as well as a comparison with other available solution approaches. Finally, Section 5 presents our concluding remarks.

2. The CC problem

2.1. Mathematical Formulation

Let $G = (V, E)$ be an undirected graph where V is the set of n vertices and E is the set of edges. In this text, a signed graph is allowed to have parallel edges but no loops. Also, we assume that parallel edges always have opposite signs. For a vertex set $S \subseteq V$, let $E[S] = \{(i,j) \in E \mid i,j \in S\}$ denote the *subset of edges induced by* S. For two vertex sets $S, W \subseteq V$, let $E[S : W] = \{(i,j) \in E \mid i \in S, j \in W\}$. One observes that, by definition, $E[S : S] = E[S]$. Consider a function $s : E \to \{+, -\}$ that assigns a sign to each edge in E. An undirected graph G together with a function s is called a *signed graph*. An edge $e \in E$ is called *negative* if $s(e) = -$ and *positive* if $s(e) = +$. Let E^- and E^+ denote, respectively, the set of negative and positive edges in a signed graph.

A *partition* of V is a division of V into non-overlapping and non-empty subsets. Consider a partition $P = \{S_1,S_2,\ldots,S_l\}$ of V. The *cut edges* and the *uncut edges* related with this partition are defined, respectively, as the edges in sets
$\cup_{1\leq i<j\leq l} E[S_i : S_j]$ and $\cup_{1\leq i\leq l} E[S_i]$. Let w_e be a nonnegative edge weight associated with edge $e \in E$. Also, for $1 \leq i, j \leq l$, let

$$\Omega^+(S_i,S_j) = \sum_{e\in E^+\cap E[S_i:S_j]} w_e \quad \text{and} \quad \Omega^-(S_i,S_j) = \sum_{e\in E^-\cap E[S_i:S_j]} w_e.$$

The *imbalance* $I(P)$ of a partition P is defined as the total weight of negative uncut edges and positive cut edges, i.e.,

$$I(P) = \sum_{1\leq i\leq l} \Omega^-(S_i,S_i) + \sum_{1\leq i<j\leq l} \Omega^+(S_i,S_j). \tag{1}$$

Likewise, the *balance* $B(P)$ of a partition P can be defined as the total weight of positive uncut edges and negative cut edges. Clearly, $B(P) + I(P) = \sum_{e\in E} w_e$. That being said, we are ready to give a formal definition to the CC problem.

Problem 2.1 (CC problem) *Let $G = (V,E,s)$ be a signed graph and w_e be a nonnegative edge weight associated with each edge $e \in E$. The correlation clustering problem is the problem of finding a partition P of V such that the imbalance $I(P)$ is minimized or, equivalently, the balance $B(P)$ is maximized.*

Observe that the given definition comprises a weighted version of the problem. To obtain a non-weighted version, it suffices to make $w_e = 1$, for each $e \in E$.

The classical mathematical formulation for the CC problem is an integer linear programming (ILP) model proposed to uncapacitated clustering problems (Mehrotra and Trick, 1998). In this formulation a binary decision variable x_{ij} is assigned to each pair of vertices $i,j \in V$, $i \neq j$, and defined as follows: $x_{ij} = 0$ if i and j are in a common set; $x_{ij} = 1$ otherwise. The model minimizes the total imbalance.

$$\text{minimize} \sum_{(i,j)\in E^-} w_{ij}(1 - x_{ij}) + \sum_{(i,j)\in E^+} w_{ij}x_{ij} \tag{2}$$

$$\text{subject to } x_{ip} + x_{pj} \geq x_{ij}, \qquad \forall\, i,p,j \in V, \tag{3}$$

$$x_{ij} = x_{ji}, \qquad \forall\, i,j \in V, \tag{4}$$

$$x_{ij} \in \{0,1\}, \qquad \forall\, i,j \in V. \tag{5}$$

The triangle inequalities (3) say that if i and p are in a same cluster as well as p and j, then vertices i and j are also in a same cluster. Constraint (4) written to $i,j \in V$ establishes that variables x_{ij} and x_{ji} assume always the same value in this formulation. Constraints (5) impose binary restrictions to the variables while the objective function (2) minimizes the total imbalance defined by equation (1). Even though this formulation is polynomial-sized, having $n(n-1)$ variables and n^3+n^2 constraints, notice that, according to constraints (4), half of the variables can be eliminated, which reduces both the number of variables and constraints of the formulation.

A set partitioning formulation (Mehrotra and Trick, 1998) is proposed in the literature to uncapacitated clustering problems and could also be used in the solution of the CC problem. As we can expect, these two formulations are not appropriate solution approaches when time limit is a constraint in the solution process. The authors in Figueiredo and Moura (2013) report that the classical formulation starts to fail (time limit set to 1h) with random instances of 40 vertices and negative density equal to 0.5.

2.2. Literature Review

To the best of our knowledge, the CC problem, as defined in the previous section, was addressed for the first time in Doreian and Mrvar (1996) (not under this name) where its heuristic solution was used as a criteria for analyzing structural balance in social networks. The heuristic approach proposed by the authors is a simple greedy neighborhood search procedure that assumes a prior knowledge of the number of clusters in the solution. This heuristic is implemented in software Pajek (Batagelj and Mrvar, 2008). Lately, motivated by the solution of a document clustering problem, the unweighted version of the CC problem was formalized in Bansal et al. (2002). The weighted version of the problem was addressed in Demaine et al. (2006). The CC problem has been largely investigated from the point of view of constant factor approximation algorithms and has been applied in the solution of many applications, including portfolio analysis in risk management (Huffner et al., 2010), biological systems (DasGupta et al., 2007), efficient document classification (Bansal et al., 2002), detection of embedded matrix structures (Gülpinar et al., 2004) and community structure (Traag and Bruggeman, 2009; Macon et al., 2012).

A comparison of several heuristic strategies (greedy and local search methods) for the problem is presented in Elsner and Schudy (2009) and applied to document clustering and natural language processing (instances of $n = 1000$), to which ILP does not scale. In this context, the authors' recommended strategy for solving the CC Problem is a greedy algorithm called $VOTE/BOEM$, which can quickly achieve good objective values with tight bounds.

In Yang et al. (2007), the CC problem is called *community mining* and an agent-based heuristic is proposed to its solution. As far as we know, there are three metaheuristic approaches applied to the CC problem. A solution based on genetic algorithms has been proposed in Zhang et al. (2008) for the CC problem and applied to document clustering, but unfortunately there is no

explanation about how the genetic operators are applied, making it difficult to understand and reproduce the proposed algorithm. Recently, Drummond et al. (2013) presented a GRASP (Feo and Resende, 1995) implementation that provides an efficient solution to the CC problem in networks of up to 8000 vertices. Later on, Levorato et al. (2014) introduced an ILS (Lourenço et al., 2003) metaheuristic for the CC problem, which outperformed, in processing time, the GRASP metaheuristic proposed earlier, with similar or improved solution quality.

3. Parallelizing local search for the CC problem in the GPU

Our work started with an analysis of two existing metaheuristics for the CC problem. Drummond et al. (2013) report the results obtained with sequential and parallel GRASP procedures. The algorithm was implemented in C++ with MPI for message passing. Then, based on this work, Levorato et al. (2014) later introduced an ILS metaheuristic for the CC problem, which was an improvement over the GRASP algorithm proposed earlier.

By observing the great amount of time spent on the local search phase of the aforementioned algorithms (Figure 1), we saw an opportunity to improve their performance by extending both of them with a new implementation of local search, which is capable of solving the problem faster, without altering the behavior of the metaheuristic. In this section we present a local search procedure for the CC problem that uses the parallelism offered by GPGPUs.

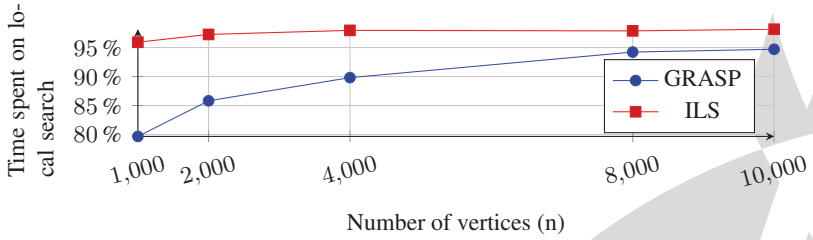

Figure 1: Time spent by GRASP and ILS on sequential 1-opt local search on Slashdot-based signed graphs.

3.1. Using General Purpose GPUs to solve optimization problems

The use of Graphics Processing Units (GPUs) has been extended to a wide range of application domains (e.g. computational science) thanks to the publication of the CUDA (Compute Unified Device Architecture) development toolkit (NVIDIA, 2015), which allows GPU programming in C-like language. When used as general-purpose computing devices, GPUs can efficiently accelerate many non-graphics programs, especially vector-and matrix-based codes that exhibit lots of parallelism with low synchronization requirements. Because their hardware is primarily designed to perform complex computations on blocks of pixels at high speed and with wide parallelism, GPU architectures differ substantially from conventional CPU hardware. Therefore, writing efficient programs to solve combinatorial optimization problems on GPUs is not a straightforward task and requires a huge effort not only at design but also at implementation level. Indeed, several challenges mainly related to the hierarchical memory management have to be dealt with. The major issues consist of efficient distribution of data processing between CPU and GPU, thread synchronization, optimization of data transfer between the different memories, as well as the capacity constraints of these memories (Van Luong et al., 2013).

Whenever parallel algorithms are applied to solve optimization problems, it is worth noticing that, in general, for distributed architectures, the global performance in metaheuristics is limited by high communication latencies. However, in GPU architectures, performance is bounded by memory access latencies. This being said, several works have already demonstrated the potential speedups when using GPUs to accelerate metaheuristics. For example, GRASP, ILS and evolutionary algorithms have already been adapted to use local search procedures implemented in GPGPU. Table 1 lists some results available in the literature.

3.2. GPGPU architecture and the CUDA programming model

CUDA has made possible the development of algorithms to solve time-consuming problems using the large number of parallel multiprocessors as well as the high memory bandwidth

Author	Main contribution	Speedup
Krüger et al. (2010)	Generic local search (memetic) algorithm	Between x70 and x120
Fujimoto and Tsutsui (2011)	Highly-parallel TSP solver for a GPU computing platform	Up to x24.2
Coelho et al. (2012)	Single Vehicle Routing Problem with Deliveries and Selective Pickups in CPU-GPU	From x2.73 to x16.23
Rocki and Suda (2012)	Accelerating TSP 2 and 3-opt local search using GPU	From x3 to x26 compared to parallel CPU w/ 32 cores
Van Luong et al. (2013)	GPU computing for parallel local search algorithms	From x0.5 up to x73.3
Pena et al. (2014)	Parallel algorithm for Siting Observers on Terrain problem	More than x20

Table 1: Speedups obtained when using GPGPUs to accelerate metaheuristics.

provided by GPUs. To accomplish high-performance computing, it is necessary to develop parallel algorithms that are partially or totally executed on the GPU. The CUDA-enabled graphics cards are composed of multiple processors, more specifically, Single Instruction Multiple Data (SIMD) processors called Stream Multiprocessors (SMs), which allow the execution of multiple parallel threads. Thus, GPU processors can efficiently execute instructions involving operations with data parallelism, when the same operation is applied to different data.

Depending on the algorithm, GPUs can provide greater processing power than CPUs because they are specialized in performing parallel tasks involving many calculations. On the other hand, CPUs are optimized for execution flow control and data cache. The physical difference between both architectures can be visualized in Figure 2: GPUs dedicate most of their area for processing units (in green), while CPUs dedicate most of their area for execution control and data cache (in yellow and orange, respectively).

A CUDA application consists in code that is executed on CPU and functions (called kernels) that are executed on GPU. The GPU is able to do parallel processing by creating threads such that each thread may execute the kernel operations on different data. This way, the GPU is used as a coprocessor to perform certain tasks more efficiently than the CPU. The GPU processing units (CUDA cores) are grouped to share a single instruction unit, so that threads mapped on these cores execute the same instruction each cycle, but on different data. Each logical group of threads sharing instructions is called a warp. Moreover, threads belonging to different warps can execute different instructions on the same cores, but in a different time slot. In practice, CUDA cores are time-shared between warps, and a group of threads in a warp performs as a SIMD unit.

Moreover, modern GPU architectures relax SIMD constraints by allowing threads in a given warp to execute different instructions (i.e. if-then-else statements and loop-termination conditions). However, these varying instructions cannot be executed concurrently, since each SIMD unit must execute the same instruction on all cores. This way, the instructions are serialized in time, which can severely degrade performance. This situation is called (thread) divergence.

Another major concern about CUDA implementation which greatly impacts performance is memory access. Bottlenecks can appear not only during data transfer between host (CPU) and device (GPU) memory, but also during memory access on the device; namely, data locality is very important. Memory requests exhibiting spatial locality are maximally coalesced. For example, accesses to addresses i and $i + 1$ are served by a single memory fetch, as long as they are aligned. Depending on the accessed addresses, concurrent memory requests from multiple threads from a warp can exhibit undesired effects. Different threads writing to the same memory address will exhibit non-deterministic behavior (it is not possible to determine which value will be actually written). Non-coalesced memory requests (including atomic ones) will be serialized in a nondeterministic order. This last behavior, often called the scattering access pattern, greatly reduces memory

throughput, since each memory request utilizes only a few bytes from each memory fetch.

The CUDA programming model includes the notion of shared memory and thread blocks, a reflection of the underlying hardware architecture as shown in Figure 3. All threads in a thread block can access the same shared memory, which provides lower latency and higher bandwidth access than global GPU memory but is limited in size. Threads in a thread block may also communicate with each other via this shared memory.

Figure 2: Basic structure of a typical CPU (left) and GPU (right).

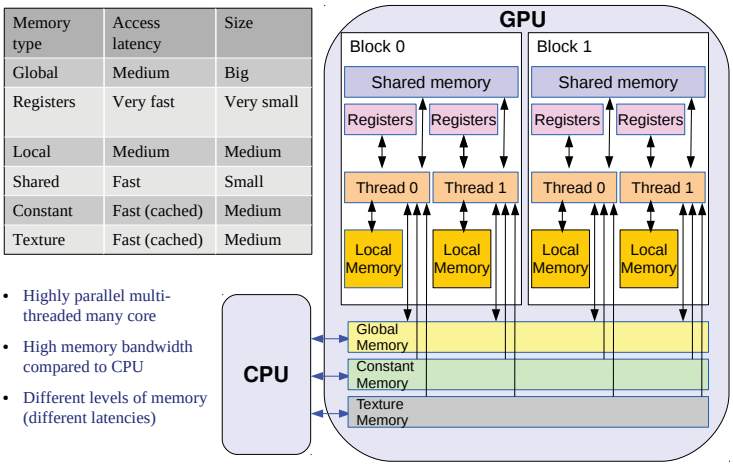

Figure 3: GPU Memory Hierarchy (Melab et al., 2011).

3.2.1. Modifying the search algorithm to run in the GPU

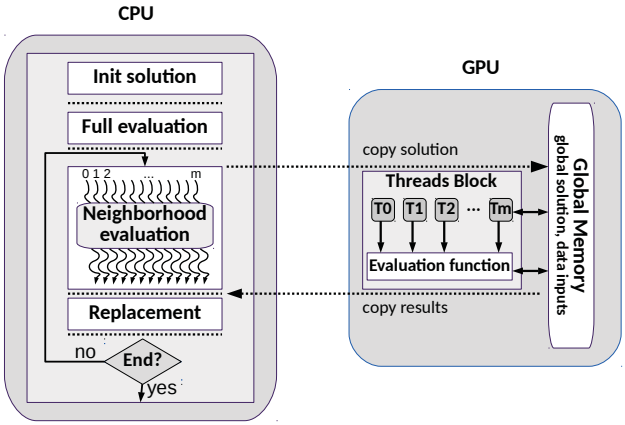

Figure 4: CUDA local search parallelization scheme (Melab et al., 2011).

Our approach to parallelize the local search procedure followed the Iteration-level Parallel Model (Van Luong et al., 2013). As can be seen on Figure 4, the evaluation of the neighborhood is made in parallel. At the beginning of each iteration, the master thread, that runs on the CPU, makes the current solution available to all threads of the GPU. Each of them evaluates a specific movement in the neighborhood of candidates, and the results are returned back to the master.

At this point, it is important to list some optimizations in the Correlation Clustering local search algorithm that have been applied for the code to run efficiently in the GPU. First of all, the graph had to be stored in Compressed Sparse Row format (Figure 5), in order to save space and

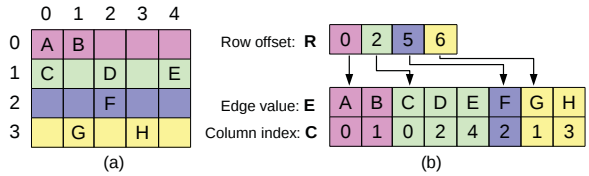

Figure 5: Using the Compressed Sparse Row (CSR) format to store graph's adjacency matrix. This representation consists of two arrays. The column indices array is a concatenation of each vertex's adjacency list into an array of m elements. The row offsets array is an $n + 1$ element array that points at where each vertex's adjacency list begins and ends within the column indices array.

avoid unnecessary data transfers between host (CPU) and device (GPU) memory. Also, since it is impossible to store the graph in shared memory (shared memory size is limited to 48kB), the graph is copied to the (slower) GPU global memory. It is then used to calculate matrices that contain the sum of edge weights between vertex i and every cluster k in the current solution. As we are processing a signed graph, there are 2 sum matrices: one for positive edges and the other for negative edges ($positiveSum$ and $negativeSum$, respectively). These matrices, also stored in GPU global memory, contain all the information needed to evaluate the imbalance of a new clustering configuration, without the need to traverse the graph, thus saving GPU memory accesses and execution time.

3.3. CUDA local search kernel implementation

Algorithm 1: $1OptLocalSearchKernel$

```
1  Input: positiveSum[], negativeSum[], cluster[], currentImbalance, number of clusters (c), vertices (n)
2  Output: destImbalance[]
3  i = idx  mod n;          → The number of vertex i is derived from each thread's unique identifier (idx)
4  k2 = idx div n;          → Vertex i is being moved to cluster k2
5  if (i ≤ n and k2 ≤ c + 1)
6     k1 = cluster[ i ];    → obtains the cluster number of vertex i
7     /* calculates only the difference in positive and negative imbalance */
8     positiveSum = - positiveSum[ i + k2 × n ] + positiveSum[ i + k1 × n ];
9     negativeSum = - negativeSum[ i + k1 × n ] + negativeSum[ i + k2 × n ];
10    destImbalance[ idx ] = currentImbalance + positiveSum + negativeSum;
```

Algorithm 1 presents the kernel pseudocode for CUDA CC 1-opt local search kernel and Figure 6 summarizes the work executed. Each thread running in the GPU (uniquely identified by idx) is responsible for calculating the delta of imbalance caused by moving a specific vertex i to a different cluster, for example, in the range k_1 to k_c. Afterwards, another kernel performs a reduction of the results, also in parallel, returning the best move for this specific local search.

Finally, whenever a vertex move is applied due to an improvement in imbalance, a third CUDA kernel is invoked to update the clustering configuration and the vertex-cluster edge-weight-sum arrays ($positiveSum$ and $negativeSum$) after a change in the clustering. This update is a necessary step to allow the execution of the Variable Neighborhood Descent procedure, that is, invoking the 1-opt local search procedure again (new local search iteration), as long as the obtained clustering solution brings an improvement in imbalance.

4. Experimental results

The algorithms described in the previous section were implemented in ANSI C++ and "C for CUDA V6.5" (NVIDIA, 2015) programming environment. All experiments were performed (with exclusive access) on a workstation with an Intel Core i7 QuadCore processor @3.40GHz (only one CPU core used), 32GB of RAM and NVIDIA Tesla K40 GPU (containing 12GB of memory and 2880 CUDA cores), under Ubuntu Linux 12.04. All heuristic outcomes are average results of 5 independent executions. Speedups are computed by dividing the sequential CPU time with the parallel time, which is obtained with the same CPU and the GPU acting as a co-processor.

Computational experiments were carried out on (i) a set of 24 random instances, and (ii) a set of 4 social networks from the literature. Next, we briefly describe these instances[1].

[1] all instances are available in http://www.ic.uff.br/∼ yuri/files/CCinst.zip.

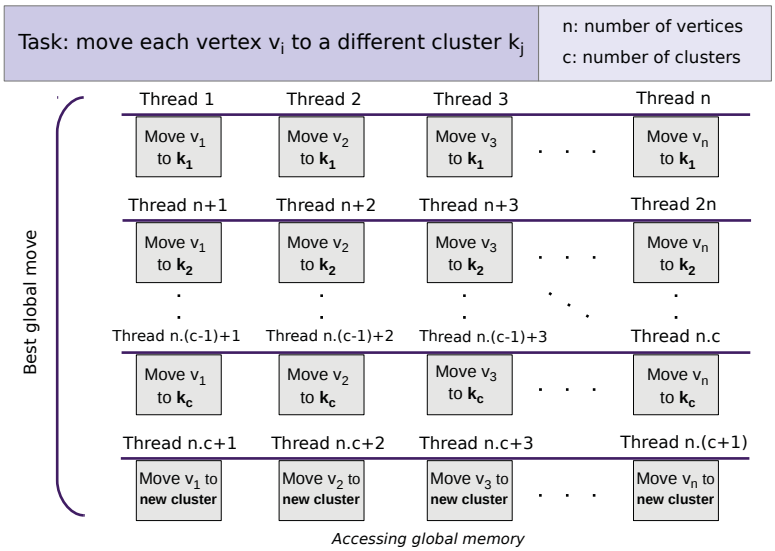

Figure 6: GPU thread work representation for 1-opt local search. Each thread idx is responsible for moving vertex i to a different cluster, from k_1 to k_c, and to a new cluster ($k_c + 1$).

(i) We generated random social networks with $n \in \{400, 600\}$, varying network density $d = 2 \times |E|/(n^2 - n)$ and negative graph density defined here as $d^- = |E^-|/|E|$. For each value of n, we considered a set of 12 random instances having d and d^- ranging, respectively, in sets $\{0.1, 0.2, 0.5, 0.8\}$ and $\{0.2, 0.5, 0.8\}$.

(ii) This set of instances is composed by 4 signed networks extracted from the large scale social network representing the technology-related news website Slashdot (Leskovec et al., 2010; Facchetti et al., 2011), containing the first n vertices, with $n \in \{2000, 4000, 8000, 10000\}$.

4.1. Sequential GRASP *vs.* Sequential GRASP with CUDA local search

In this section, we present the experiments performed with the sequential GRASP algorithm (SeqGRASP) in its best configuration, available in Drummond et al. (2013), and the sequential GRASP with CUDA Parallel Variable Neighborhood Descent (SeqGRASP/CUDAVND), when solving random instances (Table 2) and Slashdot instances (Table 3). Both experiments used the following set of parameters:

Time limit	Alpha	Neighborhood	Number of iterations without improvement
2 hours	$\alpha = 1.0$	$r = 1$	$iter = 400$

4.2. Sequential ILS *vs.* Sequential ILS with CUDA local search

Here we list the results of the experiments performed with the sequential ILS algorithm (SeqILS) in its best configuration, available in Levorato et al. (2014), and the sequential ILS with CUDA Parallel Variable Neighborhood Descent (SeqILS/CUDAVND), when solving random instances (Table 2) and Slashdot instances (Table 3). The following configuration was used in the ILS procedure.

Time limit	Alpha	Neighborhood	Iterations	ILS iterations	Perturbation level
2 hours	$\alpha = 1.0$	$r = 1$	$iter = 10$	$iterMaxILS = 5$	$perturbMax = 30$

5. Concluding remarks

The aim of this paper was to design an efficient parallelization strategy for the implementation of a parallel local search procedure for the Correlation Clustering problem on GPU. After applying the procedure, known as CUDAVND, in existing GRASP and ILS metaheuristics for the CC problem, our experimental results showed significant speedups, outperforming, in processing time, the local search available in the literature.

The GRASP/CUDAVND algorithm presented an average speedup of x5.1 (up to x14.4) on random instances and x1.71 (up to x2.01) on Slashdot instances, while the ILS/CUDAVND

n	$\|E\|$	$\|E^+\|$	$\|E^-\|$	d	d^-	**SeqGRASP**		**SeqGRASP/CUDAVND**				**SeqILS**		**SeqILS/CUDAVND**			
						Avg I(P)	**Avg Time**	**Avg I(P)**	**Gap %I(P)**	**Avg Time**	**Speedup**	**Avg I(P)**	**Avg Time**	**Avg I(P)**	**Gap %I(P)**	**Avg Time**	**Speedup**
400	15960	7980	7980	0.1	0.5	5794.8	81.22	5810.8	0.28%	16.43	4.94	5677.5	129.34	5781.2	1.83%	13.00	9.95
400	15960	3192	12768	0.1	0.8	2335.6	131.95	2362.8	1.16%	29.14	4.53	2178.4	371.80	2272.8	4.33%	11.98	31.04
400	31920	25536	6384	0.2	0.2	6384	3.39	6384.0	0.00%	4.06	0.84	6384	5.09	6384	0.00%	22.43	0.23
400	31920	15960	15960	0.2	0.5	12846.8	147.09	12841.6	-0.04%	45.29	3.25	12756.8	200.80	12844.8	0.69%	18.75	10.71
400	31920	6384	25536	0.2	0.8	5329.6	210.57	5356.8	0.51%	56.06	3.76	5145	727.50	5262	2.27%	17.78	40.92
400	79800	63840	15960	0.5	0.2	15960	5.41	15960.0	0.00%	6.36	0.85	15960	11.80	15960	0.00%	34.60	0.34
400	79800	39900	39900	0.5	0.5	34821.2	459.72	34834.8	0.04%	82.55	5.57	34672.8	535.40	34850.8	0.51%	34.95	15.32
400	79800	15960	63840	0.5	0.8	14618.8	853.29	14664.4	0.31%	115.12	7.41	14432.5	1501.91	14580	1.02%	35.78	41.98
400	127680	102144	25536	0.8	0.2	25536	7.84	25536.0	0.00%	9.10	0.86	25536	21.20	25536	0.00%	46.74	0.45
400	127680	63840	63840	0.8	0.5	57426	866.11	57432.8	0.01%	138.10	6.27	57266.8	897.89	57507.2	0.42%	51.00	17.60
400	127680	25536	102144	0.8	0.8	24072.8	1362.00	24123.6	0.21%	170.81	7.97	23888.5	2545.05	24055.2	0.70%	53.74	47.36
600	35940	28752	7188	0.1	0.2	7188	5.85	7188.0	0.00%	6.18	0.95	7188	5.81	7188	0.00%	25.81	0.23
600	35940	17970	17970	0.1	0.5	13922.8	253.88	13915.2	-0.05%	40.84	6.22	13752	329.79	13867.2	0.84%	18.09	18.23
600	35940	7188	28752	0.1	0.8	5734	645.78	5750.8	0.29%	83.92	7.70	5439.5	1322.67	5646	3.80%	16.61	79.63
600	71880	57504	14376	0.2	0.2	14376	7.09	14376.0	0.00%	7.69	0.92	14376	10.13	14376	0.00%	32.26	0.31
600	71880	35940	35940	0.2	0.5	30132	490.38	30152.4	0.07%	95.79	5.12	29964	546.51	30118.8	0.52%	27.20	20.09
600	71880	14376	57504	0.2	0.8	12609.6	1089.83	12651.2	0.33%	126.09	8.64	12316.8	2506.19	12540.8	1.82%	28.21	88.85
600	179700	143760	35940	0.5	0.2	35940	14.79	35940.0	0.00%	13.94	1.06	35940	31.44	35940	0.00%	51.31	0.61
600	179700	89850	89850	0.5	0.5	80697.2	1938.21	80670.0	-0.03%	246.26	7.87	80473	1527.27	80752.8	0.35%	53.49	28.55
600	179700	35940	143760	0.5	0.8	33763.2	3830.44	33848.8	0.25%	267.17	14.34	33478	5218.82	33743.6	0.79%	56.44	92.47
600	287520	230016	57504	0.8	0.2	57504	22.19	57504.0	0.00%	20.26	1.10	57504	63.82	57504	0.00%	70.07	0.91
600	287520	143760	143760	0.8	0.5	132066	3085.51	132176.8	0.08%	433.58	7.12	131810.4	2816.32	132238	0.32%	80.29	35.07
600	287520	57504	230016	0.8	0.8	55115.2	5777.12	55183.2	0.12%	402.15	14.37	54821.2	7134.75	55128.4	0.56%	86.39	82.59
Average						**-**	**887.19**	**-**	**0.15%**	**100.86**	**5.10**	**-**	**1186.02**	**-**	**0.87%**	**37.72**	**27.65**

Table 2: SeqGRASP, SeqGRASP/CUDAVND, SeqILS and SeqILS/CUDAVND results for random instances in (i). Number of vertices: n; Avg I(P): average value of the best solution found; AvgTime: average time spent (in seconds) on 5 executions of each algorithm. Gap %I(P) is the % gap between sequential and CUDA-based heuristics.

Instance					SeqGRASP		SeqGRASP/CUDAVND				SeqILS		SeqILS/CUDAVND			
n	$\|E^-\|$	$\|E^+\|$	$w(E^-)$	$w(E^+)$	$AvgI(P)$	$AvgTime$	$AvgI(P)$	$Gap\%I(P)$	$AvgTime$	$Speedup$	$AvgI(P)$	$AvgTime$	$AvgI(P)$	$Gap\%I(P)$	$AvgTime$	$Speedup$
2000	3217	17598	3217	17598	2186.4	136.09	2187.2	0.04%	90.18	1.51	2189.0	27.53	2188.4	-0.03%	25.09	1.10
4000	8664	40868	8664	40868	6202.0	639.30	6204.2	0.04%	325.75	1.96	6206.2	166.21	6204.4	-0.03%	42.59	3.90
8000	22789	86916	22789	86916	16084.6	3039.21	16086.7	0.01%	1513.02	2.01	16072.0	593.51	16091.8	0.12%	118.22	5.02
10000	29805	109266	29805	109266	20586.8	5615.25	20588.4	0.01%	3486.75	1.61	20598.8	1095.63	20617.2	0.09%	160.02	6.85
Average						**2357.46**		**0.02%**	**1353.93**	**1.77**		**470.72**		**0.04%**	**86.48**	**4.22**

Table 3: SeqGRASP, SeqGRASP/CUDAVND, SeqILS and SeqILS/CUDAVND results for Slashdot instances in (ii).

showed an average speedup of x28 (up to x93) on random instances and x4.2 (up to x6.9) on Slashdot instances. In both algorithms, the solution quality was equal or close to their sequential counterparts.

The next step of our work will focus on improving the analysis of larger signed social networks. The numerical experience indicates that, in order to handle instances like Epinions (131,828 vertices and 841,372 edges) or Slashdot (82,144 vertices and 549,202 edges) networks, we need to develop better parallelization strategies. One possible approach is implementing a hybrid application, using the parallelism available both in CPU (multicore) and GPU (CUDA).

References

Abell, P. and Ludwig, M. (2009). Structural balance: a dynamic perspective. *Journal of Mathematical Sociology*, 33:129–155.

Bansal, N., Blum, A., and Chawla, S. (2002). Correlation clustering. In *Proceedings of the 43rd annual IEEE symposium of foundations of computer science*, pages 238–250, Vancouver, Canada.

Batagelj, V. and Mrvar, A. (2008). Pajek wiki. http://pajek.imfm.si/. Accessed on 12.05.2014.

Cartwright, D. and Harary, F. (1956). Structural balance: A generalization of heiders theory. *Psychological Review*, 63:277–293.

Coelho, I. M., Ochi, L. S., Munhoz, P. L. A., Souza, M. J. F., Farias, R., and Bentes, C. (2012). The single vehicle routing problem with deliveries and selective pickups in a cpu-gpu heterogeneous environment. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1606–1611. IEEE Computer Society.

DasGupta, B., Encisob, G. A., Sontag, E., and Zhanga, Y. (2007). Algorithmic and complexity results for decompositions of biological networks into monotone subsystems. *BioSystems*, 90:161–178.

Demaine, E. D., Emanuel, D., Fiat, A., and Immorlica, N. (2006). Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361:172–187.

Doreian, P. and Mrvar, A. (1996). A partitioning approach to structural balance. *Social Networks*, 18:149–168.

Doreian, P. and Mrvar, A. (2009). Partitioning signed social networks. *Social Networks*, 31:1–11.

Drummond, L., Figueiredo, R., Frota, Y., and Levorato, M. (2013). Efficient solution of the correlation clustering problem: An application to structural balance. In Demey, Y. and Panetto, H., editors, *On the Move to Meaningful Internet Systems: OTM 2013 Workshops*, volume 8186 of *Lecture Notes in Computer Science*, pages 674–683. Springer Berlin Heidelberg.

Elsner, M. and Schudy, W. (2009). Bounding and comparing methods for correlation clustering beyond ilp. In *ILP'09 Proceedings of the Workshop on Integer Linear Programming for Natural Language Processing*, pages 19–27.

Facchetti, G., Iacono, G., and Altafini, C. (2011). Computing global structural balance in large-scale signed social networks. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 108, pages 20953–20958.

Feo, T. A. and Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133.

Figueiredo, R. and Frota, Y. (2014). The maximum balanced subgraph of a signed graph: Applications and solution approaches. *European Journal of Operational Research*, 236(2):473 – 487.

Figueiredo, R. and Moura, G. (2013). Mixed integer programming formulations for clustering problems related to structural balance. *Social Networks*, 35(4):639–651.

Fujimoto, N. and Tsutsui, S. (2011). A highly-parallel tsp solver for a gpu computing platform. In *Numerical Methods and Applications*, pages 264–271. Springer.

Gülpinar, N., Gutin, G., Mitra, G., and Zverovitch, A. (2004). Extracting pure network submatrices in linear programs using signed graphs. *Discrete Applied Mathematics*, 137:359–372.

Heider, F. (1946). Attitudes and cognitive organization. *Journal of Psychology*, 21:107–112.

Huffner, F., Betzler, N., and Niedermeier, R. (2010). Separator-based data reduction for signed graph balancing. *Journal of Combinatorial Optimization*, 20:335–360.

Hummon, N. P. and Doreian, P. (2003). Some dynamics of social balance processes: bringing heider back into balance theory. *Social Networks*, 25(1):17–49.

Inohara, T. (1998). On conditions for a meeting not to reach a deadlock. *Applied Mathematics and Computation*, 90:1–9.

Kim, S., Yoo, C. D., Nowozin, S., and Kohli, P. (2014). Image segmentation usinghigher-order correlation clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(9):1761–1774.

Krüger, F., Maitre, O., Jiménez, S., Baumes, L., and Collet, P. (2010). Speedups between$\times$ 70 and$\times$ 120 for a generic local search (memetic) algorithm on a single gpgpu chip. In *Applications of Evolutionary Computation*, pages 501–511. Springer.

Kunegis, J., Lommatzsch, A., and Bauckhage, C. (2009). The slashdot zoo: mining a social network with negative edges. In *WWW'09 Proceedings of the 18th international conference on World wide web*, pages 741–750.

Leskovec, J., Huttenlocher, D., and Kleinberg, J. (2010). Signed networks in social media. In *CHI'10 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1361–1370.

Levorato, M., Figueiredo, R., Drummond, L., and Frota, Y. (2014). Uma metaheurística iterated local search aplicada ao problema de correlação de clusters. In *Anais do XLVI Simpósio Brasileiro de Pesquisa Operacional (SBPO'14)*.

Lourenço, H. R., Martin, O. C., and Stützle, T. (2003). *Iterated local search*. Springer.

Macon, K., Mucha, P., and Porter, M. (2012). Community structure in the united nations general assembly. *Physica A: Statistical Mechanics and its Applications*, 391:343–361.

Mehrotra, A. and Trick, M. A. (1998). Cliques and clustering: A combinatorial approach. *Oper. Res. Lett.*, 22(1):1–12.

Melab, N., Talbi, E.-G., et al. (2011). Gpu-based multi-start local search algorithms. In *Learning and Intelligent Optimization*, pages 321–335. Springer.

NVIDIA (2015). CUDA Toolkit. http://www.nvidia.com/cuda. Accessed on 23.03.2015.

Pena, G. C., Andrade, M. V., Magalhaes, S. V., Franklin, W., and Ferreira, C. R. (2014). An improved parallel algorithm using gpu for siting observers on terrain. In *16th International Conference on Enterprise Information Systems (ICEIS-2014)*, pages 367–375.

Rocki, K. and Suda, R. (2012). Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 489–495. IEEE.

Srinivasan, A. (2011). Local balancing influences global structure in social networks. In *PNAS of the United States of America*, volume 108, pages 1751–1752.

Traag, V. and Bruggeman, J. (2009). Community detection in networks with positive and negative links. *Physical Review E*, 80:036115.

Van Luong, T., Melab, N., and Talbi, E.-G. (2013). Gpu computing for parallel local search metaheuristic algorithms. *Computers, IEEE Transactions on*, 62(1):173–185.

Yang, B., Cheung, W., and Liu, J. (2007). Community mining from signed social networks. *IEEE Transactions on Knowledge and Data Engineering*, 19:1333–1348.

Zhang, Z., Cheng, H., Chen, W., Zhang, S., and Fang, Q. (2008). Correlation clustering based on genetic algorithm for documents clustering. In *IEEE Congress on Evolutionary Computation*, pages 3193–3198.