Efficient Overlapping Document Clustering Using GPUs and Multi-core Systems

Lázaro J. González Soler, Airel Pérez-Suárez, and Leonardo Chang

Advanced Technologies Application Center (CENATAV), 7^{ma} a ♯ 21406 e/ 214 y 216, Rpto. Siboney, Playa, C.P. 12200, La Habana, Cuba
{jsoler,asuarez,lchang}@cenatav.co.cu

Abstract. Overlapping clustering algorithms have been successfully applied in several contexts. Among the reported overlapping clustering algorithms, OClustR is the one showing the best trade-of between quality of the clusters and efficiency, in the task of document clustering; however, it has a quadratic computational complexity so it could be less useful in applications dealing with a very large number of documents. In this paper, we propose two parallel versions of the OClustR algorithm, specifically tailored for GPUs and multi-core CPUs, which enhance the efficiency of OClustR in problems dealing with a very large number of documents. The experimental evaluation over standard document collections showed the correctness and good performance of our proposals.

Keywords: Data Mining, Overlapping Clustering, Parallel Algorithms.

1 Introduction

Clustering is one of the most important techniques in unsupervised learning. This technique aims at organizing a collection of objects into a set of clusters, such that objects belonging to the same cluster are more similar than objects belonging to different clusters [1]. Clustering algorithms have been successfully applied in many applications, as community detection, social networks analysis and topics detection, among others.

Since in real live an object does not need to belong to only one class, a problem that has recently received a lot of attention is the development of overlapping clustering algorithms; that is, clustering algorithms that allow the objects to belong to more than one cluster. OClustR [2] is a graph-based overlapping algorithm that has shown, in the task of documents clustering, the better trade-off between quality of the clusters and efficiency, among the existing overlapping clustering algorithms. Although OClusR attains good achievements in document clustering, it has time complexity $O(n^2)$ so it could be less useful in applications dealing with a very large number of documents. Motivated by this fact, in this work we extend the OClustR algorithm for efficiently processing very large collections of documents.

A technique that has been widely used in recent years to speed up computing tasks is parallel computing. An API for parallel programming that has gained a lot of attention with the emergence of multi core processors is OpenMP. This API provides to programmers a simple and flexible interface for developing parallel applications on any platform and it expresses parallelism in multi-core shared-memory systems. Another technology that has increased the computing power of many applications in recent years is the use of Graphics Processing Units (GPU). A GPU is a device that was initially designed for processing algorithms belonging to the graphical world. However, due to the low cost, the high level of parallelism and the optimized floating-point operations GPUs offer, they have been used in several applications of different kinds.

The main contribution of this paper is the design and implementation of two parallel versions of the OClustR algorithm, specifically tailored for GPUs and multi-core CPUs, respectively. The proposed parallel versions enhance the efficiency of OClustR in problems dealing with a very large number of documents, like for instance news analysis, information organization and profiles identification, among others.

The remainder of this paper is organized as follow: in Section 2, we present a brief description of the OClustR algorithm. In Section 3, we introduce two parallel versions of OClustR. The experimental evaluation, showing the performance of our proposed parallel algorithms, is presented in Section 4. Finally, conclusions and some ideas about future work are presented in Section 5.

2 The OClustR Algorithm

Let $O = \{o_1, o_2, \ldots, o_n\}$ be a collection of objects, $\beta \in [0, 1]$ a similarity threshold and S a symmetric similarity function between the objects of O.

OClustR builds an overlapping clustering in three steps [2]. In the first step, OClustR represents the collection as a *weighted thresholded similarity graph* $\widetilde{G}_\beta = \langle V, \widetilde{E}_\beta, S\rangle$. This graph meets that $V = O$ and there is an edge $(v, u) \in \widetilde{E}_\beta$ iff $S(v, u) \geq \beta$; each edge $(v, u) \in \widetilde{E}_\beta, v \neq u$ is labeled with the value of $S(v, u)$. This first step has a computational complexity of $O(n^2)$. Once $\widetilde{G}_\beta$ is built, in the second step OClustR builds an initial set of clusters through a covering of $\widetilde{G}_\beta$, using *weighted star-shaped sub-graphs* (ws-graphs). A ws-graph in $\widetilde{G}_\beta$, denoted by $G^\star = \langle V^\star, E^\star, S\rangle$, is a sub-graph of $\widetilde{G}_\beta$, having a vertex $c \in V^\star$, such that there is an edge between c and all the other vertices in $V^\star \setminus \{c\}$. The vertex c is called the *center* of the ws-graph and the remaining vertices are called *satellites*. Isolated vertices are considered *degenerated* ws-graphs.

OClustR transforms the problem of building the covering of $\widetilde{G}_\beta$ using ws-graphs into the problem of selecting a list C of vertices, each one being the center of a ws-graph of the cover. For building list C, OClustR first builds a list L containing the vertices of $\widetilde{G}_\beta$ with *relevance* greater than zero; isolated vertices are directly included in C. The relevance of a vertex v is computed as the average of the *relative density* and the *relative compactness* of v. Te relative density of v is computed as the ratio between the number of vertices $u \in v.Adj$ having $|v.Adj| \geq |u.Adj|$ and the size of $v.Adj$. The relative compactness of v is computed as the ratio between the number of vertices $u \in v.Adj$ having

$AIS(v) \geq AIS(u)$ and the size of $v.Adj$, where $AIS(v)$ is the average similarity existing between v and the satellites of the ws-graph determined by v; $AIS(u)$ is defined in a similar way to $AIS(v)$.

Once L is built, it is sorted in descending order of the relevance of the vertices. After that, OClustR visits each vertex $v \in L$ and if v is not covered yet or if v is already covered but it has at least one adjacent vertex u which is not covered yet, then v is considered as a center and consequently, it is added to C. Each selected vertex, together with its adjacent vertices, constitutes a cluster in the initial set of clusters. This process of selection has time complexity $O(n^2)$.

Finally, in the third step, OClustR processes C in order to remove those vertices forming a *not useful* cluster. A vertex v forms a not useful cluster if the ws-graph it determines shares more vertices with others selected ws-graphs than the ones it only contains. For removing those vertices, the list C is ordered in descending order of the number of adjacent vertices of the vertices included in C and, after that, each vertex $v \in C$ is visited in order to remove from $v.Adj$ any vertex $u \in C$ forming a not useful cluster. If a vertex $u \in v.Adj$ forms a not useful cluster, it is removed from C and the satellites it only covers are included in the cluster determined by v. This steps also has a computational complexity of $O(n^2)$. After this process, the remaining clusters constitute the final clustering.

3 Proposed Algorithms

In this section, we introduce two parallel versions of OClustR. The first version, called OMP-Clus, implements OClustR on OpenMP, while the second one, called CUDA-Clus, implements it on CUDA.

3.1 OMP-Clus Algorithm

Although the three steps used by OClustR for building the clustering have time complexity $O(n^2)$, the one that consumes more processing time is the first one; that is, the construction of $\widetilde{G}_\beta$. Therefore, in order to speed up the OClustR algorithm, first of all we should speed up the construction of this graph.

For building $\widetilde{G}_\beta = \langle V, \widetilde{E}_\beta, S \rangle$, the similarity between every pair of vertices must be computed; thus, for speeding up the construction of $\widetilde{G}_\beta$ the similarity of a vertex wrt. the remaining vertices of the graph is computed in parallel. Since the similarity function used for building $\widetilde{G}_\beta$ is symmetric, the similarity between a specific pair of vertices v, u has to be computed only one time. Therefore, assuming that V is a list containing the vertices of the $\widetilde{G}_\beta$, in order to build the edges relatives to a vertex $v \in V$ we only need to compute the similarity between v and the vertices that come after it in V.

Let $P = \{P_1, P_2, \ldots, P_k\}$ be a set of k processors on a computer, v a vertex of $\widetilde{G}_\beta$, and Suc_v the set of vertices that come after v in V. In order to parallelize the computation of the similarity between a vertex v and each vertex of Suc_v, we apply a *work-sharing* technique. Work-sharing is a way of expressing parallel execution and it provides replicated execution of the same code segment on multiple threads. This way, the vertices on Suc_v are partitioned into k subsets, each one containing $\frac{|Suc_v|}{k}$ vertices, and each of these subsets is assigned to a processor in P. Then, each processor $P_i \in P$ will compute the similarity between v and the subset of Suc_v assigned to it, hereinafter referred to as $P_i(Suc_v)$. Since each $P_i \in P$ is computing at the same time the similarity between v and a vertex of Suc_v, it could happen that more than one processor tries to modify the adjacency list of v; therefore, the modification of this list was set as a *critical region* directive, in order to avoid *race conditions*. The use of critical region was also applied for the updating of $AIS(v)$. Once all the processors have finished its work, $AIS(v)$ is updated by dividing it by $|v.Adj|$.

The second step in OClustR is to build a cover of $\widetilde{G}_\beta$. With this purpose, the relevance of the vertices must be computed in order to build the list L. The relevance of a vertex $v \in V$ depends only on its adjacent vertices; therefore, we can simultaneously compute the relevance of all the vertices. For doing this, OMP-Clus partitions the set of vertices into k subsets, each one of size $\frac{|v|}{k}$, and it assigns each one of these subsets to a processor in P. Then, a processor $P_i \in P$ computes, one by one, the relevance of each vertex v belonging to its assigned subset. With this aim, P_i visits each vertex $u \in v.Adj$ and it adds $\frac{1}{|v.Adj|}$ to the relevance of v if $|v.Adj| \geq |u.Adj|$ and it also adds $\frac{1}{|v.Adj|}$ to the relevance of v if $AIS(v) \geq AIS(u)$. Once P_i has computed the relevance of a vertex v, like OClustR does, it this relevance is greater than zero, it adds v to L. On the other hand, if v is isolated, P_i adds v to C. Since more than one processor could tries to add a vertex to L or C at the same time, the modification of both lists was also set as a *critical region* directive.

Analyzing the selection strategy used by OClustR, we can conclude that the processing of two vertices of L that belongs to different connected components does not affect the way in which C is built neither the way C is then post-processed in order to remove vertices forming not useful clusters. Let $M = \{M_1, M_2, \ldots, M_T\}$ be the set of connected components of $\widetilde{G}_\beta$. Based on the aforementioned idea, OMP-Clus partitions M into k subsets of size $\frac{T}{k}$ and it assigns each one of these subsets to a processor $P_i \in P$. Each processor $P_i \in P$ will apply over each one of its assigned components, exactly the same selecting and filtering strategy used by OClustR.

3.2 CUDA-Clus Algorithm

OClustR has several steps that spent a high number of operations. A massively parallel implementation of OClustR in CUDA should take advantage of the benefits of GPUs, like for instance, the high bandwidth communication between CPU and GPU, and the GPU memory hierarchy. In this section, we present a massively parallel implementation in CUDA of OClustR, namely CUDA-Clus, that speeds up the most time-consuming phases of OClustR, *i.e.*, the construction of $\widetilde{G}_\beta$, and the computation of the relevance of the vertices. The other steps of OClustR were not implemented in CUDA because they are high memory-consuming tasks which are more favored in a CPU implementation.

Let $D = \{d_1, d_2 \ldots, d_n\}$ be a collection of documents described by a set of terms. Let $T = \{t_1, t_2, \ldots, t_m\}$ be the set containing all the different terms that describe at least one document in D. CUDA-Clus, like OMP-Clus, assumes that a document $d_i \in D$ is represented by two vectors, one containing the positions that the terms describing d_i have in T, and the other one containing the weights those terms have in the description of d_i. For computing the similarity between two documents, CUDA-Clus uses the cosine measure [3], which has been the most used for this purpose. The cosine measure between two documents d_i, d_j is defined as the ratio between $\sum_{k=1}^{m} d_i(k) * d_j(k)$ and $\|d_i\| \cdot \|d_j\|$.

To speed up the construction of $\widetilde{G}_\beta$ CUDA-Clus computes in parallel the similarity between each $d_i \in D$ and each one of the documents in Suc_{d_i}. However, CUDA-Clus goes beyond and it also parallelizes the computation of the similarity between a pair of documents, in order to speed up even more the construction of $\tilde{G}_\beta$. From the definition of the cosine measure, it can be seen that its numerator is a sum of several independent products; thus, CUDA-Clus takes advantage from this fact and it performs these products all at once. Since the norm of a document can be computed while the document is being read, CUDA-Clus does not spend an extra time for computing the denominator of the cosine measure.

In order to carry out the previous idea, CUDA-Clus builds a grid comprised of $k = \frac{n \cdot (t-1)}{t}$ squared blocks, each block having a shared memory square matrix (SMM); where t is the dimension of both the blocks and the matrices, and it is the maximum value allowed by the architecture of the GPU for the dimension of a SMM. The use of SMM and the low latency it has allows CUDA-Clus to do not constantly access to the CPU memory and to speed up the aforementioned calculus. When CUDA-Clus is computing the similarity between a document d_i and the documents in Suc_{d_i}, the set Suc_{d_i} is partitioned in k subsets and each one of these subsets is assigned to a block comprising the grid. In this context, a column of a block represents a terms vector associated to a document assigned to that block and each row on that column represents the weight of a term in the description of the document. Each cell of the blocks forming the grid has a thread that performs a set of operations. In our case, each thread associated with a column inside a block will compute the similarity between d_i and the document associated with that column.

Based on the aforementioned memory organization, for computing the similarity between d_i and the document associated with the jth column inside a block, hereinafter referred to as d_j, each thread on that column will compute the product between the weight of the term associated with that thread, with the weight that has the same term in d_i and the result is stored on the SMM associated with the block. If a document assigned to that block has more than t terms a technique named *Tiling* is applied. Tiling consists on dividing a set of data into a number of small subsets in such a way that each of these subsets fits into the SMM [4]. Thus, based on this technique, CUDA-Clus iteratively moves the threads of a column toward the next subset of unprocessed terms and it reuses the information previously stored on the SMM. Once all the products have been computed a *Reduction* is applied over each column of the SMM, to compute in parallel the sum of all the values on a column [4]. The sum obtained by applying Reduction on a column corresponds with the numerator obtained when the cosine measure is applied between d_i and the document represented by that column. This sum is then normalized and copied to the CPU. During the above mentioned steps the $AIS(d_i)$ can also be computed, without affecting the computational cost of those steps.

Once $\widetilde{G}_\beta$ has been built, CUDA-Clus paralellizes the computation of the relevance of each vertex, following a similar reasoning to the one employed for building $\widetilde{G}_\beta$. CUDA-Clus uses the grid for computing at the same time the relevance of all the vertices. With this aim, the set of vertices is partitioned into k subsets and each one of these subsets is assigned to a block comprising the grid. Each column inside a block represents a vertex $v_i \in V$ and each row in that column represents an adjacent vertex of v_i. Different from building $\widetilde{G}_\beta$, a column inside a block will compute the relevance of the vertex it represents. Each thread on a column determines the contribution of the corresponding adjacent vertex to the relevance of the vertex represented by the column. A vertex u, represented by a thread on the column representing vertex v, contributes $\frac{1}{|v.Adj|}$ to the relevance of v if $|v.Adj| \geq |u.Adj|$; otherwise, its contribution is zero. On the other hand, u contributes too $\frac{1}{|v.Adj|}$ to the relevance of v if $AIS(v) \geq AIS(u)$; otherwise, its contribution is zero. The total contribution provided by a vertex associated with a thread is stored in the SMM of the block. If a vertex assigned to that block has more than t adjacent vertices the *Tiling* technique is applied over the column. Once the threads on each column finished its work, a *Reduction* is applied over each column of the SMM in order to compute the relevance of the associated vertices and then, the results are copied to the CPU. The remaining steps of CUDA-Clus are the same as in OClustR.

4 Experimental Results

In this section, the results of several experiments, testing the performance of the proposed algorithms, are presented. The experiments were conducted over five documents collections and focused on: (1) to assess the correctness of the proposed implementations and (2) to evaluate the speed-up achieved by the proposed algorithms wrt. the original OClustR [2]. All the algorithms were implemented in C++; the code of OClustR algorithm was obtained from their authors. For implementing CUDA-Clus the CUDA toolkit 5.5 was used. All the experiments were performed on a PC with Core i7-4770 processor at 3.40 GHz, 8GB RAM, having a PCI express NVIDA GeForce GT 635, with 1 GB DRAM.

The document collections used in our experiments were built from three benchmark text collections commonly used in document clustering: TDT2, Reuters-21578 and Reuters-v2. Both Reuters-21578 and Reuters-v2 can be obtained from http://kdd.ics.uci.edu, while the TDT2 benchmark can be obtained from http://www.nist.gov/speech/tests/tdt.html. From these benchmarks five document collections were built. The characteristics of these collections are shown in Table 1.

Table 1. Overview of the collections used in our experiments

Characteristics/Collection	Reu	Reuter	Reu-v2	TDT	Reu-all
#Documents	11 367	15 017	18 562	16 006	79 514
#Terms	27 083	100 814	29 150	68 023	95 530
Terms/Docs	45	58	52	144	101

In our experiments, documents were represented using the Vector Space Model (VSM) [5]. The index terms of the documents represent the lemmas of the words occurring at least once in the collection; stop words were removed. The index terms of each document were statistically weighted using their frequency.

As it was mentioned before, the first experiment assesses the correctness of the proposed parallel versions. With this aim, we compare the clustering results of OClustR on the collections Reu and TDT using $\beta = 0.20$ and $\beta = 0.25$, with those obtained by OMP-Clus and CUDA-Clus under the same conditions. Results were compared using the FBcubed external evaluation measure [6], where the OClustR results were used as *ground truth.* This measures takes values in [0,1], where 1 means identical results and 0 completely different results. The average FBcubed value obtained by both OMP-Clus and CUDA-Clus was 0.995, with a standard deviation of 0.00568 and 0.00602, respectively. We think that the differences between the clusterings are caused by the inherent data order dependency of OClustR and also because the floating point arithmetic used by CUDA. From this experiment, we can conclude that the speed-up attained by each proposed algorithm does not degrade their accuracy.

In the second experiment, we compare the algorithms regarding to the time they spent for processing each of the experimental collections. With this aim, each algorithm was executed over each collection, using $\beta = 0.15, 0.25$ and 0.35. Since OClustR as well as the two parallel versions depend on the data order analysis, experiments were repeated five times, for each collection-parameter combination, varying the order of the documents and the average time spent was computed. Table 2 shows the average time, in seconds, spent by each algorithm for processing each collection, for each β value; for OMP-Clus and CUDA-Clus algorithms the speed-up attained wrt. the time spent by OClustR is also showed.

Table 2. Performance of each algorithm over the collections, for each β value

β	Algorithms	**Reu**	**TDT**	**Reu-v2**	**Reu-all**	**Reuter**
0.15	OClustR	202.9	1041	710.3	33744.6	1261.4
	OMP-Clus	82/**2.6x**	367/**2.9x**	694/**1.1x**	12953.5/**2.6x**	366.8/**3.4x**
	CUDA-Clus	19.5/**10.5x**	92.5/**11.3x**	111.2/**6.4x**	1145.7/**33.5x**	43.1/**29.5x**
0.25	OClustR	202.9	205.2	611.6	36129.6	1284.3
	OMP-Clus	65.4/**3.1x**	302.4/**3.4x**	229/**2.7x**	9214/**3.9x**	354.6/**3.6x**
	CUDA-Clus	16.3/**12.6x**	80.2/**12.9x**	55.6/**11x**	838.6/**43.1x**	30/**42.4x**
0.35	OClustR	210.1	1110.3	616.4	34044.1	1743.4
	OMP-Clus	67.2/**3.1x**	310.5/**3.6x**	293.6/**2.1x**	8940.7/**3.8x**	363.3/**4.8x**
	CUDA-Clus	15.3/**13.8x**	72.3/**15.4x**	34.1/**18.1x**	636.3/**53.5x**	24.6/**70.9x**

As it can be seen in Table 2, our proposals achieve considerable speed-up rates wrt. the time spent by OClustR, in all the experimental collections, being CUDA-Clus the best. In order to validate the improvement attained by our proposals, the statistical significance of the above results was verified, using the Mann-Whitney test, with a 95% of confidence. As a result of this test, it has been verified that the results obtained by CUDA-Clus are statistically superior to those obtained by the OClustR algorithm, whilst the results of OMP-Clus they are not. It is important to remember that these experiments were performed over a PCI express NVIDA GeForce GT 635, which has only two streaming processors. Thence, we expected that if we increase the number of streaming processors of the GPU, the speed-up of our algorithm will be higher.

5 Conclusions

In this paper, we introduced two parallel versions of the OClustR for document clustering, called OMP-Clust and CUDA-Clus, using GPUs and Multi-core CPUs, respectively. The experimental evaluation conducted over standard document collections showed that the speed-up attained by our parallel versions does not degrade their accuracy. These experiments also showed that our proposals achieve considerable speed-up rates, being the results obtained by CUDA-Clus statistically superior to those obtained by OClustR. From these results, we can conclude that CUDA-Clus is suitable for problems dealing with a very large number of documents, like for instance news analysis, information organization and profiles identification, among others. As future work we are going to explore the use of the texture memory in the implementation of CUDA-Clus.

References

1. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. 31(3), 264–323 (1999)
2. Pérez-Suárez, A., Martínez-Trinidad, J.F., Carrasco-Ochoa, J.A., Medina Pagola, J.E.: A New Graph-based Algorithm for Overlapping Clustering. Neurocomputing 121, 234–247 (2013)
3. Berry, M.: Survey of Text Mining, Clustering, Classification and Retrieval. Springer (2004)
4. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)
5. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM 18(11), 613–620 (1975)
6. Amigó, E., Gonzalo, J., Artiles, J., Verdejo, F.: A comparison of extrinsic clustering evaluation metrics based on formal constraints. Information Retrieval 12, 461–486 (2009)