Parallel Subspace Clustering Using Multi-core and Many-core Architectures

Amitava Datta[1], Amardeep Kaur[1], Tobias Lauer[2](✉), and Sami Chabbouh[2]

[1] School of Computer Science and Software Engineering, University of Western Australia, Perth, Australia
[2] Department of Electrical Engineering and Information Technology, Offenburg University of Applied Sciences, Offenburg, Germany
tobias.lauer@hs-offenburg.de

Abstract. Finding clusters in high dimensional data is a challenging research problem. Subspace clustering algorithms aim to find clusters in all possible subspaces of the dataset where, a subspace is the subset of dimensions of the data. But exponential increase in the number of subspaces with the dimensionality of data renders most of the algorithms inefficient as well as ineffective. Moreover, these algorithms have ingrained data dependency in the clustering process, thus, parallelization becomes difficult and inefficient. SUBSCALE is a recent subspace clustering algorithm which is scalable with the dimensions and contains independent processing steps which can be exploited through parallelism. In this paper, we aim to leverage, firstly, the computational power of widely available multi-core processors to improve the runtime performance of the SUBSCALE algorithm. The experimental evaluation has shown linear speedup. Secondly, we are developing an approach using graphics processing units (GPUs) for fine-grained data parallelism to accelerate the computation further. First tests of the GPU implementation show very promising results.

Keywords: Data mining · Subspace clustering · Multi-core architectures · Many-core architectures · GPU computing

1 Introduction

The growing size and dimensions of data these days have set new challenges for data mining research. Clustering is a data mining process of grouping similar data points into clusters without any prior knowledge of the underlying data distribution. Due to the curse of dimensionality, data group together differently under different subsets of dimensions, called subspaces. Subspace clustering algorithms attempt to find clusters in all possible subsets of dimensions of a given data set [1,2].

The area of subspace clustering is of critical importance with diverse applications [1]. But due to the exponential search space with the increase in dimensions, subspace clustering becomes computationally very expensive. With the wide availability of multi-core processors and the spread of many-core coprocessors such as GPUs, parallelization seems to be an obvious choice to reduce this computational cost.

SUBSCALE is a recent subspace clustering algorithm to find the subspace clusters without enumerating the data points or computing any redundant clusters [3,4]. This algorithm combines the dense set of points across all single dimensions of the data to find non-trivial subspace clusters. Although SUBSCALE algorithm scales well with the dimensions and performs faster than other subspace clustering algorithms, it is still compute intensive due to the generation of combinatorial 1-dimensional dense set of points. However, the compute time can be reduced by parallelizing the computation of the dense units.

In this paper, we aim to parallelize the SUBSCALE algorithm in two ways and investigate its runtime performance. First, we exploit current multi-core architectures with up to 48 processing cores using OpenMP. The experimental evaluation demonstrates the speedup of up to a factor of 27. This modified algorithm is faster and scalable for high dimensional large data sets. Second, we use many-core graphics processing units to exploit data parallelism on a fine-granular level, with a significant speedup, especially for larger computations. The latter work is ongoing and results are expected to further improve with optimizations in the implementation.

In the next section we discuss the current related literature. Section 3 explains subspace clustering and the algorithm we parallelize. In Sect. 4, we describe our multi-core parallelization and analyse the performance of the parallel implementation. Section 5 describes our current work of massive parallelization using GPUs with preliminary results. The paper is concluded in Sect. 6.

2 Related Work

Over the past few years, there has been extensive research on clustering algorithms [2]. The underlying premise that data group together differently under different subsets of dimensions opened the challenging domain of subspace clustering algorithms [1,5].

However, the increase in the dimensions of data impedes the performance of clustering algorithms which are known to perform very well with low dimensions. Moreover, most of the subspace clustering algorithms have less obvious parallel structures [6,7]. This is partially due to the data dependency during the processing sequence [8].

SUBSCALE [3,4] is a recent subspace clustering algorithm and requires only k database scans to process a k-dimensional dataset. Also, this algorithm is scalable with dimensions and its structure contains the computation of independent tasks which can be parallelized. In the next section, we briefly discuss the SUBSCALE algorithm and our modifications for multi-core parallel implementation.

3 Subspace Clustering

This section provides the basics and definitions of subspace slustering and a brief description of SUBSCALE [4], the algorithm which we aim to make more efficient through parallelization.

Given an $n \times k$ set of data points, a point P_i is a k-dimensional vector $\{P_i^1, P_i^2, \ldots, P_i^k\}$ such that, P_i^d is the projection of a point P_i on the d^{th} dimension. A *subspace* is a subset of k dimensions. A subspace cluster $C_i = (P, S)$ is a set of points P, such that the projections of these points in subspace S, are dense.

According to the Apriori principle [6], a dense set of points in a subspace S of dimensionality a, is dense in all of 2^a projections of S. Thus, it is sufficient to find a cluster in its maximal subspace. A cluster $C_i = (P, S)$ is called a **maximal subspace cluster**, if there is no other cluster $C_j = (P, S')$ such that $S' \supset S$. The SUBSCALE Algorithm finds such maximal subspace clusters by combining the dense points from single dimensions and without computing the redundant non-maximal clusters.

3.1 SUBSCALE Algorithm

The main idea behind the SUBSCALE algorithm is to find the dense sets of points (also called *density chunks*) in all of the k single dimensions, generate the relevant *signatures* from these density chunks, and *collide* them in a hash table ($hTable$) to directly compute the maximal subspace clusters as explained below.

Density Chunks. Based on two user defined parameters ϵ and τ, a data point is *dense* if it has atleast τ points within ϵ distance. The neighbourhood $N(P_i)$ of a point P_i in a particular dimension d is a set of all the points P_j such that $L_1(P_i^d, P_j^d) < \epsilon$, $P_i \neq P_j$. L_1 is the distance metric. Each dense point along with its neighbours, forms a *density chunk* such that each member of this chunk is within ϵ distance from each other.

The smallest possible dense set of points is of size $\tau + 1$, called a *dense unit.* In a particular dimension, a density chunk of size t can have $\binom{t}{\tau+1}$ possible combinations to form the dense units. Some of these dense units may or may not contain projections of the higher dimensional maximal subspace clusters. Without any prior information of the underlying data distribution, it is not possible to know the promising dense units in advance. Only viable solution is to check which of these dense units from different dimensions contain identical points.

Signatures. The SUBSCALE algorithm proposed a novel way to match the dense units by assigning signatures to them. To create signatures, each of the n data points is mapped to a random, unique and large integer key. The sum of the mapped keys of the data points in each dense unit is termed as its *signature.*

According to observations 2 and 3 in the SUBSCALE paper [3], *two dense units with equal signatures would have identical points in them with extremely high probability.* Thus, collisions of the signatures across dimensions $d_r, \dots, d_s$ implies that, the corresponding dense unit exists in the maximal subspace, $S = \{d_r, \dots, d_s\}$. We refer our readers to the extended version of the original paper [4] for the detailed explanation. Each single dimension may have zero or more dense chunks, which in turn generate different number of signatures in each dimension. Some of these signatures will collide with the signatures from the other dimensions to give a set of dense points in the maximal subspace.

Hash Table. The SUBSCALE algorithm uses a hash table data structure *hTable* to store collision information about each signature. An *hTable* has a fixed number of slots and each slot can store one or more signatures, depending upon the implementation (Fig. 1). When a signature *Sig* is generated in a dimension d, it is mapped to a slot in the *hTable*. In this paper, we used *modulo numSlots* for mapping of signatures to a slot. If a slot already contains a signature Sig' such that $Sig = Sig'$, then d is appended to the dimension-list attached to *Sig*.

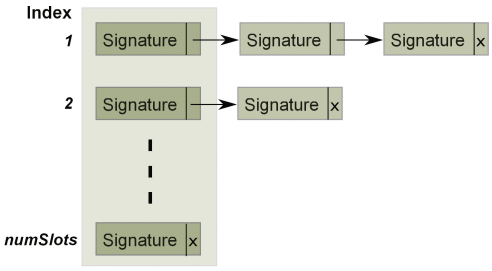


Fig. 1. *hTable* data structure.

Since the size of each dense unit is $\tau + 1$, the value of a signature generated from a dense unit will lie in the range $R = [(\tau+1) \cdot min_K, (\tau+1) \cdot max_K]$, where min_K and max_K are the smallest and the largest keys respectively. Also, if $numSig_d$ is the number of total signatures in a dimension d, then the total number of signatures in a k-dimensional data set will be $total_{sig} = \sum_{d=1}^{k} numSig_d$. If memory is no onstraint a hash table with $|R|$ slots can easily accommodate $total_{sig}$, as typically, $total_{sig} \ll R$. Since memory is a constraint, the range R can be split into multiple slices and each slice can be processed independently using a separate and smaller hash table. The computations for each slice is not dependent of other slices. The split factor called *sp* determines the number of splits of R and its value can be set according to the available working memory. Also, the cluster quality is not affected by splitting of hash table computations. The clusters are formed by combining the dense units in each maximal subspace. The total number of dense units are decided by the density chunks created through epsilon neighbourhoods in single dimensions. As long as all dense units are processed, same clusters will be generated through sequential or parallel methods.

4 Multi-core Parallelization Using OpenMP

We used the OpenMP platform with C to parallelize the SUBSCALE algorithm. OpenMP is a set of complier directives and callable runtime library routines to facilitate shared-memory parallelism [9].

4.1 Processing Dimensions in Parallel

The generation of signatures from the density chunks in each single dimension is independent of other dimensions. Thus, the dimensions can be divided among the available processing cores to be run in parallel using threads. The hash table *hTable* is shared among threads. However, the problem of thread contention arises when multiple threads try to get mutually exclusive access of the same slot of *hTable* to update or store the signature information. Without mutually exclusive access, two threads with the same signatures generated from two different dimensions, would overwrite the same slot of *hTable*. The maximal subspace of a dense unit can only be found by having the information about which of the dimensions generated this dense unit. We discuss the results from this method in Sect. 4.3.

4.2 Processing Slices in Parallel

The other approach to avoid thread contention is to utilise the splitting of the range R of expected signature values as proposed by the SUBSCALE algorithm. The slices created through the splitting can be processed in parallel as each slice generates signatures from different range compared to other slices. Each slice requires a separate hash table. Though this approach helps to achieve faster clustering performance from the SUBSCALE algorithm, the memory required to store all of the hash tables can still be a constraint. Since R denotes the whole range of computation sums that are expected during the signature generation process, we can bring these slices into the main working memory one by one. Each slice is again split into sub-slices to be processed with multiple threads. The total number of signatures can be pre-calculated from the dense chunks in all dimensions. The results and their evaluation are discussed in the next section.

4.3 Results and Analysis

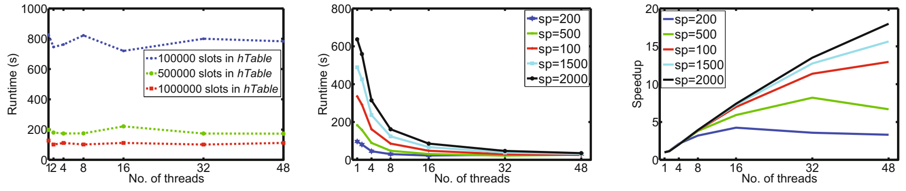
The experiments were carried out on the IBM Softlayer Server Quad Intel Xeon E7-4850, 2 GHz, with 48 cores, 128 GB RAM and Ubuntu 15.04 kernel. Hyper-threading was disabled on the server so that each thread could run on a separate physical core and parallel performance could be measured fairly. The parallel version of the SUBSCALE algorithm was implemented in C using OpenMP directives. Also, we used 14-digit non-negative integers for the key database.

The two main datasets for this experiment: 4400×500 *madelon* dataset [10] and 3661×6144 *pedestrian* dataset [11,12], are publicly available. These datasets were also used by authors of the SUBSCALE algorithm.

Multiple Cores for Dimensions. We used the *madelon* data set with $\epsilon = 0.000001$, $\tau = 3$ and experimented with three different number of slots in the shared *hTable*: 0.1 million, 0.5 million and 1 million. Figure 2a shows the runtime performance of the *madelon* data set by using multiple threads for dimensions. We observe that performance improves slightly by processing dimensions in parallel but as discussed before, thread contention due to mutually exclusive access to the same slot in the shared hash table results in performance degradation.

Multiple Cores for Slices. To avoid this memory contention due to shared *hTable*, we split the hash table computations into slices according to the SUBSCALE algorithm and distribute these slices among multiple cores. Figure 2b shows the results of the runtime versus the number of threads used for processing the slices of the *madelon* dataset. The hash computation was sliced with different values of split factor *sp* ranging between 200 and 2000. We can see the performance boost by using more threads. The speed up is shown in Fig. 2c, which becomes linear as the number of slices increases.

Scalability with Dimensions. The 6144 dimensional *pedestrian* dataset is used to study the speed up with the increase in dimensions. With $\epsilon = 0.000001$ and $\tau = 3$, 19860542724 total signatures are expected which would require $\sim$ 592 GB of working memory to store the hash tables. To overcome this huge memory requirement, we can split these signature computations twice. We used a split factor of 60 to bring down the memory requirement for total hash tables. Each of these 60 slices were further split into 200 subslices to be run on 48 cores.



Dimensions in parallel Slices in parallel Speed up

Fig. 2. *madelon* dataset: $\epsilon = 0.000001$ and $\tau = 3$. In (a), the total dimensions are divided among available cores using threads. Due to thread contention, the runtime fails to improve. In (b), the slices of hash computation (*sp* denotes the split factor) are distributed among multiple cores and runtime improves with number of threads.

The number of slots in each *hTable* are fixed using $\frac{total\_signatures}{sp}$. The execution time decreases drastically with increase in the number of threads. It took around 26 h to finish processing all of the 60×200 slices. The sequential version of SUBSCALE clustering algorithm takes about ~ 720 h.

5 Fine-Grained Parallelization Using GPUs

In this section, we describe an alternative way of parallelization with a much finer-grained task structure, suitable for parallelization on graphics processing units (GPUs). This work is ongoing, the results are preliminary.

5.1 Levels of Granularity

Finer-grained parallelism than the one described so far can be achieved on two levels of granularity. First, we observe that within a dimension all density chunks can be processed independently, since they are necessarily disjoint. Hence, if a dimension contains k dense chunks, k independent tasks can be executed in parallel by k threads. The downside of this approach is that the number and sizes of density chunks in a dimension is not known a priori, thus resulting in a vastly varying number of active threads with varying workloads during the process, which is not efficient and not very suitable for GPU parallelization.

However, we note that the processing within a single density chunk can also be parallelized. Recall that this computation consists of computing the signatures of all dense units, i.e. all possible combinations of $\tau + 1$ elements, in that chunk. Signature computation only requires read access to the points in the respective dense unit, so even for non-disjoint units there are no thread conflicts. Furthermore, since all the dense units of the same dimension result in different signatures (with very high probability, cf. Sect. 3.1), hash collisions are very unlikely during the parallel computation within one dimension. Hence, one notable advantage of this task structure is that there will be no thread contention for accessing the hash table, if different dimensions are processed sequentially. (Note, however, that this does not preclude parallel computation of dimensions.)

5.2 Parallel Task Structure

In this parallel approach, a task consists of computing the signature value for one single dense unit and hashing it. All such tasks can be executed in parallel.

```
for each dense unit du[i] of length tau in parallel
    signature := 0
    for j = 0 to tau
         signature := signature + key(du[i].point[j])
    htable.insert(signature)
```

Algorithm 1. Code for parallel computation of signatures

Since a density chunk of t elements has $\binom{t}{\tau+1}$ dense units, this approach results in a large number of small and almost identical tasks. As τ is constant within one execution of the algorithm and since there are no branches, all tasks execute the same sequence of instructions, but on different data. This type of data parallelism is well-suited for implementation using GPUs. We have developed an implementation using Nvidia's CUDA architecture and programming model [13]. This model enables data parallelism by allowing scalable grids of threads, depending on the size of the data to be processed. Each thread is identified by its ID and can use this ID, e.g. to determine memory locations for reading input and writing output data. In our case, the ID is required to identify the dense unit to process.

5.3 Computing Subsets Efficiently

Algorithm 1 presupposes that the thread with ID i knows how to retrieve the i-th dense unit, i.e. the subset $\{P_{i_0},, P_{i_\tau}\}$ of projected points whose signature it computes. In a sequential scenario, this is not a problem, as the subsets of size $\tau + 1$ can be enumerated one after another, using an ordering in which it is computationally cheap to calculate the lexicographically next subset from a given one [14]. In our parallel scenario, however, each thread needs to identify its relevant subset independently, without reference to other results, i.e., the i-th thread must find the i-th subset without access to the $(i-1)$-th subset. Calculating directly the i-th subset is significantly more complex than advancing to the next subset from a given one. Using the combinatorial representation (or, *combinadics*) of index i allows for a relatively efficient computation of the corresponding subset [15], but still involves $O(\tau \cdot \log t)$ calculations of binomial coefficients. Using a table of pre-calculated binomial coefficients can improve efficiency at the cost of extra memory usage.

An alternative solution – which is used in our current implementation – is precomputing an array containing the lexicographic enumeration of all $\binom{t}{\tau+1}$ dense units within a density chunk of size t, i.e. the i-th position of the array contains a representation of the i-th dense unit. A straightforward and space-efficient encoding of subsets of size $\tau + 1$ of a set with t elements is a bit string of length t with exactly $\tau + 1$ bits set to 1. The precomputation of the array is sequential but uses a very efficient implementation to compute the lexicographically next bit permutation [16]. Calculating the dense unit array of length ~ 500000 for a density chunk of size $t = 60$ and dense units of size $\tau + 1 = 4$ takes about 12 ms on an Intel Core i7-4720HQ @2.6 GHz. Computing an array of ~ 50 million permutations $(t = 60, \tau + 1 = 6)$ takes 1252 ms.

We are also working on a possible parallelization of this precomputation, similar to the idea of parallel prefix calculation [17].

5.4 GPU-Based Hashing

Hashing the calculated signatures into *htable* can also be carried out on the GPU. GPU-based hashing has been extensively studied by Alcantara, who proposed several efficient hashing schemes [18]. Our approach, based on the implementation used in [19], is currently being implemented and hence, not part of the evaluation presented here. Note that GPU memory is a limiting factor for the hash table size. State of the art GPUs come with up to 16 GB of RAM, which is sufficient to accommodate each partial table of the slicing approach described in Sect. 4.2.

5.5 Performance Evaluation

Our current implementation of the GPU approach is a first step. It has not been optimized regarding the GPU's memory hierarchy and hence does not benefit from caching effects. Also, it does currently not use more intricate CUDA functions such as, for instance, the SHFL (shuffle) command, which might be interesting for combinatoric tasks like subset enumeration.

The performance of the GPU algorithm was tested on an Intel Core i7-4720HQ @2.6 GHz machine equipped with an Nvidia GeForce GTX 950M GPU hosting 640 processing units (CUDA cores) and 4 GB of GPU RAM, against the sequential CPU algorithm for computing signatures, run on the same machine. The results are shown in Table 1. They do not include the time for precomputation of subsets and for hashing the signatures, but include all transfer times between GPU and host memory required for the GPU computations.

Table 1. Performance of CPU and GPU algorithms for computing signatures.

#Signatures computed	Time CPU (ms)	Time GPU (ms)	Speedup factor
1,770	0.4	1.0	0.4
34,220	11.5	1.9	6.1
487,635	148.8	13.8	10.8
5,461,512	1770.0	135.8	13.0
50,063,860	17692.5	1162.2	15.2

For smaller numbers of signatures, GPU is slower than CPU. This was to be expected as there is always a small but non-negligible ramp-up cost for GPU kernels Note that the speedup factor increases with the amount of calculations. Note that the GPU used for this preliminary evaluation is a relatively small model; high-performance Tesla GPUs contain thousands of CUDA cores and achieve significantly higher processing power.

6 Conclusion

In this paper, we have presented two independent ways of parallelizing the SUBSCALE algorithm. First, we have described the use of a common shared memory multi-core architecture. Achieving the parallelization by assigning CPU cores to slices of the hash table for store signatures and finding larger-dimensional dense units, the results have shown linear speedup with the number of cores.

The second approach uses finer-granular data parallelism and can be implemented efficiently on graphics processing units (GPUs). First performance tests show very promising results, especially for larger data sets. This part of the work is ongoing; we are currently implementing the full functionality including GPU-based parallel hashing of the signatures.

The two approaches do not exclude each other. In fact, they can complement each other, using multi-core parallelism for coarse-grained tasks (processing of dimensions or has table slices) and many-core data parallelism for finer-grained subtasks (such as individual signature computation). Future work includes this combination of both approaches, making the best possible use of the different processing resources.

References

1. Parsons, L., Haque, E., Liu, H.: Subspace clustering for high dimensional data: a review. ACM SIGKDD Explor. Newsl. **6**(1), 90–105 (2004)
2. Aggarwal, C.C., Reddy, C.K.: Data Clustering: Algorithms and Applications, 1st edn. Chapman & Hall/CRC, Boca Raton (2013)
3. Kaur, A., Datta, A.: Subscale: fast and scalable subspace clustering for high dimensional data. In: 2014 IEEE International Conference on Data Mining Workshop (ICDMW), pp. 621–628 (2014)
4. Kaur, A., Datta, A.: A novel algorithm for fast and scalable subspace clustering of high-dimensional data. J. Big Data **2**(1), 17 (2015)
5. Sim, K., Gopalkrishnan, V., Zimek, A., Cong, G.: A survey on enhanced subspace clustering. Data Min. Knowl. Disc. **26**(2), 332–397 (2013)
6. Agrawal, R., Gehrke, J., Gunopulos, D.: Automatic subspace clustering of high dimensional data for data mining applications. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 94–105 (1998)
7. Kailing, K., Kriegel, H.P., Kroger, P.: Density-connected subspace clustering for high-dimensional data. In: SIAM International Conference on Data Mining, pp. 246–256 (2004)
8. Zhu, B., Mara, A., Mozo, A.: CLUS: parallel subspace clustering algorithm on spark. In: Morzy, T., Valduriez, P., Bellatreche, L. (eds.) ADBIS 2015. CCIS, vol. 539, pp. 175–185. Springer, Cham (2015). doi:10.1007/978-3-319-23201-0_20
9. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**, 46–55 (1998)
10. Bache, K., Lichman, M.: UCI Machine Learning Repository (2013)
11. Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision meets robotics: the KITTI dataset. Int. J. Rob. Res. **32**(11), 1231–1237 (2013)
12. Zhu, J., Liao, S., Lei, Z., Yi, D., Li, S.Z.: Pedestrian attribute classification in surveillance: database and evaluation. In: ICCV Workshop on Large-Scale Video Search and Mining (LSVSM 2013), Sydney (2013)
13. Nvidia: CUDA home page. http://www.nvidia.com/object/cuda_home_new.html. Accessed 26 May 2017

14. Loughry, J., van Hemert, J., Schoofs, L.: Efficiently enumerating the subsets of a set (2000). applied-math.org/subset.pdf
15. McCaffrey, J.: Generating the mth lexicographical element of a mathematical combination. MSDN Library (2004)
16. Anderson, S.E.: Bit Twiddling Hacks compute the lexicographically next bit permutation. http://graphics.stanford.edu/~seander/bithacks.html#NextBitPermutation. Accessed 26 May 2017
17. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. GPU gems **3**(39), 851–876 (2007)
18. Alcantara, D.A.F.: Efficient hash tables on the GPU. Ph.D. thesis, University of California Davis (2011)
19. Strohm, P.T., Wittmer, S., Haberstroh, A., Lauer, T.: GPU-accelerated quantification filters for analytical queries in multidimensional databases. In: Bassiliades, N., Ivanovic, M., Kon-Popovska, M., Manolopoulos, Y., Palpanas, T., Trajcevski, G., Vakali, A. (eds.) New Trends in Database and Information Systems II. AISC, vol. 312, pp. 229–242. Springer, Cham (2015). doi:10.1007/978-3-319-10518-5_18