Int. J. Appl. Math. Comput. Sci., 2019, Vol. 29, No. 1, 81–91
DOI: 10.2478/amcs-2019-0006

# EXPLOITING MULTI–CORE AND MANY–CORE PARALLELISM FOR SUBSPACE CLUSTERING

AMITAVA DATTA [a], AMARDEEP KAUR [a], TOBIAS LAUER [b,*], SAMI CHABBOUH [b]

[a] School of Computer Science and Software Engineering
University of Western Australia, 35 Stirling Highway, Crawley, Perth, WA 6009, Australia

[b] Department of Electrical Engineering and Information Technology
Offenburg University of Applied Sciences, Badstr. 24, 77652 Offenburg, Germany
e-mail: tobias.lauer@hs-offenburg.de

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 a subset of dimensions of the data. But the 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, which means that 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 the computational power of widely available multi-core processors to improve the runtime performance of the SUBSCALE algorithm. The experimental evaluation shows linear speedup. Moreover, we develop 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, many-core, GPU computing.

## 1. Introduction

The unprecendented growth in the size and dimensions of data has set new challenges for data mining research (Fan *et al.*, 2014). Clustering is a data mining process of grouping similar data points into clusters without prior knowledge of the underlying data distribution. Due to the curse of dimensionality, data points lose contrast in high-dimensional space, making it difficult to cluster data based on similarity measures (Steinbach *et al.*, 2004; Aggarwal and Reddy, 2013). Traditional clustering algorithms like k-means (MacQueen, 1967) and DBSCAN (Ester *et al.*, 1996) fail to find meaningful clusters in high-dimensional data.

Different combinations of data dimensions reveal different groupings among the data. These possible subsets of dimensions of the data are called subspaces. Subspace clustering algorithms attempt to find clusters in all possible subsets of data dimensions (Parsons *et al.*, 2004; Aggarwal and Reddy, 2013).

[*] Corresponding author

The area of subspace clustering is of critical importance in diverse applications (Li *et al.*, 2004; Jun *et al.*, 2006; Tierney *et al.*, 2014). However, with the increase in data dimensions, the number of subspaces increases exponentially, which makes the subspace clustering process computationally very expensive. With the wide availability of multi-core processors and the spread of many-core co-processors such as GPUs, parallelization seems to be an obvious choice to reduce computation time.

Most of subspace clustering algorithms are inefficient for high-dimensional data because they use enumeration of data points and compute redundant clusters during the clustering process. SUBSCALE (Kaur and Datta, 2015) is a recent subspace clustering algorithm which can deal efficiently with the exponential search space of high-dimensional data without enumerating data points or generating trivial clusters.

The core of the SUBSCALE algorithm resides on generation of 1-dimensional combinatorial units of dense points and combining them to form non-trivial subspace clusters. Although SUBSCALE scales well with the dimensions and performs faster than other subspace clustering algorithms, it is still computationally intensive due to the generation of combinatorial dense units from data. The computing time can be further reduced by parallelizing the computation of these dense units.

In this paper, which is an extended version of our earlier work (Datta *et al.*, 2017), we aim to parallelize the SUBSCALE algorithm in two ways and investigate the runtime performances. First, we exploit current multi-core architectures with up to 48 processing cores using the OpenMP framework. The experimental evaluation demonstrates a speedup of up to a factor of 27. Compared with the original SUBSCALE algorithm, this modified parallel algorithm is faster and more scalable for high-dimensional data sets.

Second, we use many-core graphics processing units (GPUs) to exploit data parallelism on a fine-granular level, with a significant speedup, especially for large-scale computations. We also describe an efficient combinatorial solution to the problem of assigning tasks to threads. This work is ongoing and results are expected to improve with further optimizations in its implementation.

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

## 2. Related work

The clustering problem has been studied extensively in different disciplines, including statistics (Fukunaga, 1990), image processing (Elhamifar and Vidal, 2013), bioinformatics (Thalamuthu *et al.*, 2006) and data mining (Han *et al.*, 2011). In fact, a search for ‘data clustering’ on Google Scholar (Google Scholar, 2018) found more than $3.5$ million entries in 2018. Despite the ubiquitous importance of clustering, we are still far from those perfect clustering algorithms that can work with the high-dimensional data being produced these days.

There are a number of surveys available on clustering algorithms along the time-line of their development in history (Murtagh, 1983; Jain and Dubes, 1988; Jain *et al.*, 1999; Xu and Wunsch, 2005; Parsons *et al.*, 2004; Berkhin, 2006; Kriegel *et al.*, 2009; Sim *et al.*, 2013; Aggarwal and Reddy, 2013; Xu and Tian, 2015). The clustering algorithms which are time tested and known to perform very well for low-dimensional data are not suitable for high-dimensional data due to the curse of dimensionality.

There have been efforts to reduce the data dimensionality by transforming them to new lower dimensions (Joliffe, 2002), but it is difficult to interpret the clusters found in the new dimensions in relation to the original data space. Also, these methods fail to capture the local relevance and groupings among the data. Another approach found in the literature for clustering is to use selective dimensions and numbers of clusters (Aggarwal *et al.*, 1999), but these algorithms are essentially data partitioning techniques and do not find all natural groupings of data.

Due to the curse of dimensionality, data group together differently under different subsets of dimensions, and this premise has opened the challenging domain of subspace clustering for data mining researchers (Agrawal *et al.*, 1998; Parsons *et al.*, 2004; Kailing *et al.*, 2004; Sim *et al.*, 2013; Xu and Tian, 2015). Subspace clustering algorithms aim to find all possible clusters in all possible subspaces of the data without any prior knowledge of the underlying data distribution.

CLIQUE (Agrawal *et al.*, 1998) is a famous subspace clustering algorithm which finds lower-dimensional candidate clusters through a fixed-size grid and then combines them together iteratively for computing higher dimensional clusters. There are many variations of this algorithm, for example, using entropy (Cheng *et al.*, 1999) and an adaptive grid (Nagesh *et al.*, 2001).

The increase in the dimensions of data impedes the performance of clustering algorithms. The exponential search space and the presence of redundant clusters in the subspace hierarchy of high-dimensional data add to the computational inefficiency. However, the detection of trivial clusters and an excessive number of database scans during the clustering process is inbuilt in most of subspace clustering algorithms. Moreover, most of the subspace clustering algorithms do not have obvious parallel structures, which hinders any attempt to use parallel computation for further optimization of efficiency (Zhu *et al.*, 2015). This is partially due to the data dependency during the processing sequence of cluster generation.

SUBSCALE (Kaur and Data, 2015; 2014) is a recent subspace clustering algorithm which requires only $k$ database scans to process a $k$-dimensional dataset. Also, this algorithm is scalable with the number of dimensions and its structure contains the computation of independent tasks which can be parallelized.

In the next sections, we briefly discuss the SUBSCALE algorithm and our modifications for multi-core and many-core parallel implementations.

## 3. Subspace clustering

This section provides the basics and definitions of subspace clustering and a brief description of

SUBSCALE (Kaur and Datta, 2015), 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 $\{P_i^1, P_i^2, \dots, P_i^k\}$, where $P_i^d$ is the projection of a point $P_i$ on the $d$-th dimension. A *subspace* is a subset of $k$ dimensions. A data point from a $k$-dimensional subspace can be projected on up to $2^k$ subspaces.

A *subspace cluster*, $C_i : (P, S)$, is a set $P$ of points, such that the projections of these points in a subspace $S$ are dense. According to the Apriori principle (Agrawal *et al.*, 1998), a dense set of points in a subspace $S$ of dimensionality $d$ is dense in all of $2^d$ projections of $S$. Therefore, 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 $S'$ is a superset of subspace $S$. The projections of $C_i$ in all lower-dimensional subsets of $S$ will be trivial clusters and will not add to the information that is already in cluster $C_i$. The SUBSCALE algorithm finds such maximal subspace clusters by combining dense points from single dimensions and without computing the redundant non-maximal clusters.

**3.1. SUBSCALE algorithm.** The main idea behind SUBSCALE is to find the dense sets of points (also called *density chunks*) in each single dimension. The algorithm then generates the relevant *signatures* from these density chunks. The signatures are collided with each other in a hash table ($hTable$, explained later) to directly compute the maximal subspace clusters.

**3.1.1. Density chunks.** Based on two user defined parameters $\epsilon$ and $\tau$, a data point is *dense* if it has at least $\tau$ points within $\epsilon$-distance. The neighbourhood $N(P_i)$ of a point $P_i$ in a particular dimension $d$ is the set of all points $P_j$ such that $L_p(P_i^d, P_j^d) < \epsilon$, $P_i \neq P_j$.

A density chunk is a set of one or more dense points such that each point is within $\epsilon$-distance from any other point in the chunk. Figure 1 shows an example of computing density chunks with $\tau = 3$ from sorted data points: $\{P_1, P_7, P_3, P_{12}, P_5, P_4, P_9, P_2, P_6, \dots\}$ in a particular dimension.

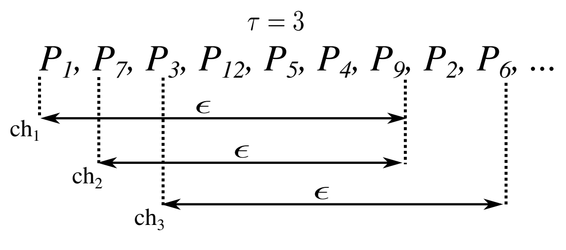

Fig. 1. Computation of density chunks.

The smallest possible dense set of points, called a *dense unit*, is of size $\tau + 1$. A density chunk of size $t$ has $\binom{t}{\tau+1}$ possible combinations to form dense units.

**Computing density chunks.** We notice in Fig. 1 that consecutive density chunks may overlap, i.e., they have common points. Density chunk $ch_2$ will not generate any further dense units than those already computed from chunk $ch_1$. Thus, a density chunk can be eliminated if its last element is the same as that of the previous chunk. Also, $\binom{5}{4} = 5$ combinations of points from chunk $ch_3$ would have already been generated by $ch_1$.

To eliminate the redundant dense unit computations due to overlapping density chunks, we can use a special marker in each density chunk called $pivot$, which is the position of the last element of the previous chunk in the current density chunk. In Fig. 1, the pivot is the last point in $ch_2$ and the fifth data point in $ch_3$. Instead of computing $\binom{7}{4} = 35$ combinations from $ch_3$, we can create partial combinations from two partitions of $ch_3$, as shown in Fig. 2. However, all the combinations from a density chunk must be computed when $pivot \leq \tau$. The pivot based approach results in considerable savings in computational time and efficiency for larger density chunks with lots of overlapping points.

After computing the dense units, the next step is to find which of these units 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. The only viable solution is to check which of the dense units from different dimensions contain identical points, which is done using signatures.

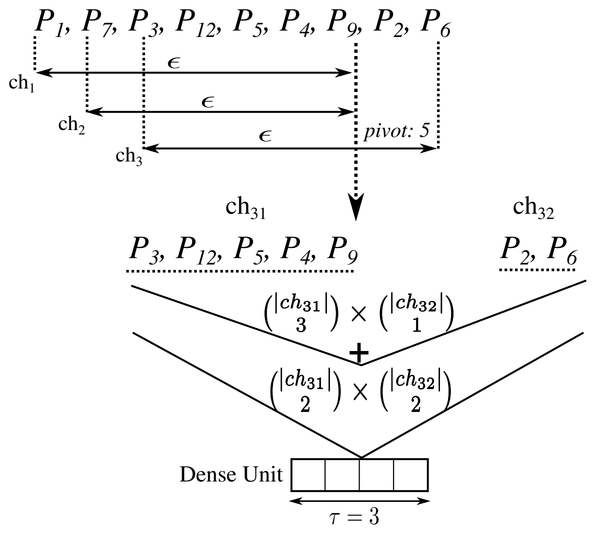
P1, P7, P3, P12, P5, P4, P9, P2, P6
ϵ
ϵ
ϵ
ch1
ch2
ch3
pivot: 5
ch31
ch32
P3, P12, P5, P4, P9
P2, P6
$\binom{|ch_{31}|}{3} \times \binom{|ch_{32}|}{1}$
+
$\binom{|ch_{31}|}{2} \times \binom{|ch_{32}|}{2}$
Dense Unit
τ = 3

Fig. 2. Optimizing the computation of density chunks; $|ch_{31}|$ and $|ch_{32}|$ are the sizes of chunks $ch_{31}$ and $ch_{32}$, respectively.

### 3.1.2. Signatures.

SUBSCALE 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 data points in each dense unit creates a *signature* for this dense unit. $Sig_m^d$ denotes the signature of dense unit $m$ in dimension $d$.

According to two observations in the SUBSCALE paper (Kaur and Datta, 2014), *two dense units with equal signatures would have identical points in them with extremely high probability*. Thus, collisions of a signature across dimensions $d_r, \ldots, d_s$ imply that the corresponding dense unit exists in the maximal subspace $S = \{d_r, \ldots, d_s\}$. We refer our readers to the extended version of the original paper (Kaur and Datta, 2015) for a detailed explanation.

Each single dimension may have zero or more density chunks, which in turn generate a different number of signatures in each dimension. Some of these signatures will collide with the signatures from other dimensions to give a set of dense points in the maximal subspace. If $Sig_m^j$ collides with another signature $Sig_n^k$ then the corresponding dense units are the same (with very high probability) and exist in subspace $\{j, k\}$.

### 3.1.3. Hash table.

SUBSCALE uses a hash table data structure $hTable$ to store collision information about each signature $Sig$ (Fig. 3). An $hTable$ has a number of slots and each slot can store one or more signatures.

When a signature $Sig$ is generated in a dimension $d$, it is mapped to a slot in $hTable$. Identical signatures should collide in the same slot and only one of them is stored, as they contain the same points. Information about the dimensions from which the signatures were generated is also stored alongside the signature in the slot.

Matching items by hash collisions is also used in relational hash joins. However, our approach is not based on relational databases. There is no distinction between small and big relations or between a partitioning/build phase and a probe phase in our work. Most importantly, our approach "joins" up to thousands of dimensions.

Here $numSlots$ is the number of slots in a hash table which can vary depending upon the implementation. In this paper, we used $modulo\ numSlots$ for the mapping of signatures to a slot (Fig. 4). If a slot already contains a signature $Sig'$ such that $Sig = Sig'$, then $d$ is appended to the dimension-list attached to $Sig$.

In order to identify all signature collisions, a sufficiently large $hTable$ is required to hold them in the working memory of the system. If $numSig_d$ is the total number of signatures in dimension $d$, then the total number of signatures in a $k$-dimensional data set will be $totalSig = \sum_{d=1}^{k} numSig_d$. As each dense unit contains $\tau + 1$ data points and the signature equals the sum of mapped keys, the value of a signature will always lie within 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.

If memory is not a constraint, a hash table with $|R|$ slots can easily accommodate $totalSig$, as typically, $totalSig \ll R$. Since memory is a constraint, the range $R$ can be split into multiple slices and only the signatures whose values fit within a slice are hashed and the rest are discarded in each iteration. Another tweak to this could be to abandon the signature generation process from a dense unit, as soon as the mapped key or the sum of mapped keys exceeds the upper boundary of its slice. This way, each slice can be processed independently using a separate and a smaller hash table.

The computations for each slice is not dependent on 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 is decided by the

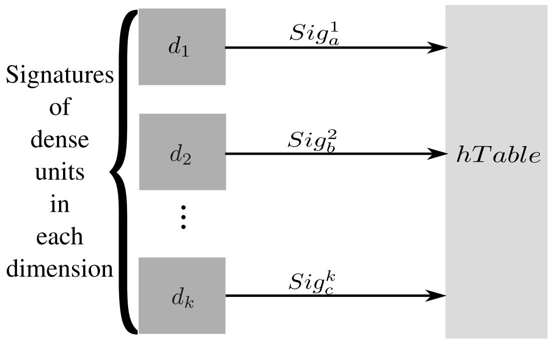

Fig. 3. Hash table data structure $hTable$ in SUBSCALE to store signatures and their associated data from multiple dimensions.

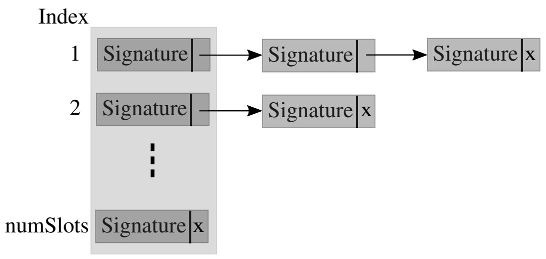

Fig. 4. Using slots in $hTable$. $numSlots$ is the total number of slots available in $hTable$. Each slot may receive 0 or more signature nodes through the $modulo$ function on the signature value.

density chunks created through $\epsilon$-neighbourhoods in single dimensions.

As long as all dense units are processed, the same clusters will be generated through sequential or parallel methods. The computations of dense units in each single dimension as well as each single slice can be processed independent of others.

In the next section, we endeavour to use this independence among dense units to reduce the execution time for the SUBSCALE algorithm with multiple cores.

## 4. Multi-core parallelization using OpenMP

OpenMP is a set of compiler directives and callable runtime library routines to facilitate shared-memory parallelism (Dagum and Menon, 1998). We used the OpenMP platform with C to parallelize SUBSCALE.

**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 (Fig. 5). The hash table $hTable$ is shared among threads. The modified SUBSCALE algorithm to process the dimensions in parallel is given in Algorithm 1.

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 mutual exclusion, two threads with the same signatures generated from two different dimensions would overwrite the same slot of $hTable$. The overwriting would lead to loss of information on the maximal subspace related to this signature. The maximal subspace of a dense unit can only be found by knowing the underlying dimensions generating its signature.

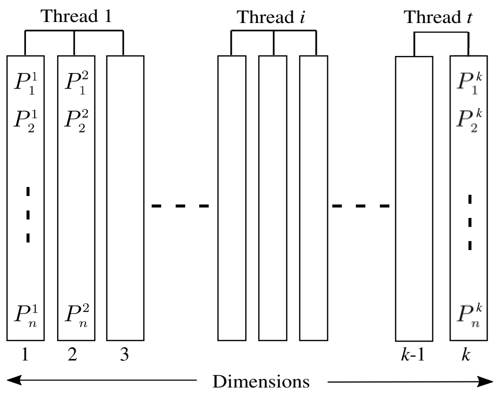

Fig. 5. Parallel processing of the SUBSCALE algorithm. Each dimension is allocated a separate thread and each thread computes the density chunks and its signatures independent of the others.

**Algorithm 1.** Modified SUBSCALE algorithm to execute multiple dimensions on multiple cores and with a shared hash table.

```
Require: $\mathbf{DB}$ : $n \times k$ data, K: set of $n$ keys
1: Initialize a common hash table $hTable$
2: Start threads with shared $\mathbf{DB}$, $hTable$ and $K$. Share the outer $for$ loop among threads
3: for dimension $j \leftarrow 1$ to $k$ do
4:     Scan $\{P_1^j, P_2^j, \ldots, P_n^j\}$ and find density chunks
5:     for each density chunk do
6:         Create signatures and hash them to $hTable$ in a mutually exclusive manner
7:     end for
8: end for
9: Synchronization barrier for threads
10: Collect all collisions from $hTable$ to output maximal dense units
11: return Dense points in maximal subspaces
```

The number of signatures being mapped to the same slot depends on their sum value, $numSlots$ in the $hTable$ and the hashing function ($modulo$ in this paper). A smaller hash table would lead to frequent requests for exclusive access to the same slot from different threads. It can be argued that a bigger hash table would result in a decrease in thread contention, but the allowed total size of a hash table depends on the available working memory. OpenMP provides a lock mechanism for the shared variables, and the number of locks to be maintained grow proportional to the slots in the hash table, so their synchronization adds to the overhead as well. We discuss the results of this approach in Section 4.3.

**4.2. Processing slices in parallel.** The other approach to avoid thread contention is to utilize the splitting of the range $R$ of expected signature values, as described in Section 3.1.3. The slices created through the splitting can be processed in parallel as each slice generates signatures from a different range compared to other slices. Each slice requires a separate hash table. The modified SUBSCALE algorithm to process the slices in parallel is given in Algorithm 2.

Though this approach helps to achieve faster clustering performance of SUBSCALE, 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

**Algorithm 2.** Modified SUBSCALE algorithm to execute multiple slices on multiple cores and separate $hTable$.

```
Require: DB : $n \times k$ data, K: set of $n$ keys, SP: split
    factor
 1: $R \leftarrow (max(K) - min(K)) \times (\tau + 1)$
 2: $SLICE = R/SP$
 3: Start threads with shared DB and $K$. Share the outer
    $for$ loop among threads
 4: for $split \leftarrow 0$ to $SP - 1$ do
 5:     Initialize a new $hTable$
 6:     $LOW = min(K) \times (\tau + 1) + split \times SLICE$
 7:     $HIGH = LOW + SLICE$
 8:     for dimension $j \leftarrow 1$ to $k$ do
 9:         Scan $\{P_1^j, P_2^j, \dots, P_n^j\}$ and find density chunks
10:         for each density chunk do
11:             create signatures and hash them to $hTable$ if
                they fall between $LOW$ and $HIGH$
12:         end for
13:     end for
14:     Collect all collisions from $hTable$ to output
        maximal dense units
15:     Discard $hTable$
16: end for
17: return Dense points in maximal subspaces
```

density 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 an IBM Softlayer Server Quad Intel Xeon E7-4850, 2 GHz, with 48 cores, 128 GB RAM and Ubuntu 15.04. 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.

We used publicly available *madelon* ($4400 \times 500$) and *pedestrian* ($3661 \times 6144$) datasets (Geiger *et al.*, 2013; Zhu *et al.*, 2013; Lichman, 2013). These datasets were also used by the authors of the SUBSCALE algorithm.

**4.3.1. 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 6 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.

**4.3.2. 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 7 shows the results of runtime versus 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 speedup is shown in Fig. 8, and becomes linear as the number of slices increases. The speedup is the time ratio between sequential algorithm and using multi-threading.

**4.3.3. Scalability with dimensions.** The 6144 dimensional $pedestrian$ dataset is used to study the speedup with the increase in dimensions. With $\epsilon = 0.000001$ and $\tau = 3$, a total of 19860542724 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 was further split into 200 sub-slices to be run on 48 cores. The number of

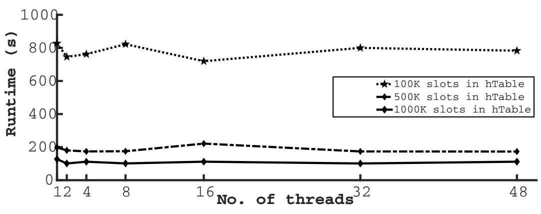

Fig. 6. Multiple cores for dimensions: runtime performance versus the number of threads for the $madelon$ dataset ($\epsilon = 0.000001$ and $\tau = 3$). The total dimensions are divided among available cores using threads. Due to thread contention, the runtime fails to improve.

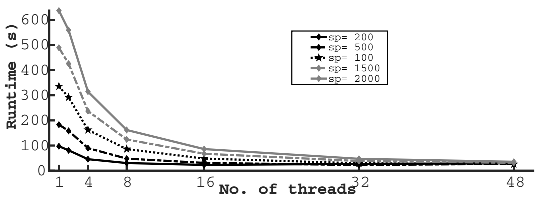

Fig. 7. Multiple cores for slices: runtime performance versus the number of threads for the $madelon$ dataset ($\epsilon = 0.000001$ and $\tau = 3$). The slices of hash computation ($sp$ denotes the split factor) are distributed among multiple cores and runtime improves with the number of threads.

slots in each $hTable$ is fixed as $total\_signatures/sp$.

The execution time decreases drastically with an increase in the number of threads. It took around 26 hours to finish processing all of the $60 \times 200$ slices. The sequential version of SUBSCALE takes $\sim 720$ hours.

## 5. Fine-grained parallelization using GPUs

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

**5.1. Level of granularity.** Recall that the main workload of the computation in SUBSCALE is the generation and hashing of signatures for the extremely large number of dense units that may occur in a single dimension. For each density chunk, the computation consists in generating the signatures of all dense units, i.e., all possible combinations of $\tau+1$ elements, in that chunk. Since a density chunk of $t$ elements consists of $\binom{t}{\tau+1}$ dense units, we get a combinatorial explosion for larger $t$, resulting in a huge number of small and almost identical tasks, independent of each other. Signature computation only requires read access to the points in the respective dense unit. So, even for non-disjoint dense units, a parallel computation will not create any thread conflicts.

Furthermore, since all the dense units of the same dimension result in different signatures (with very high probability; cf. Section 3.1.2), it is very unlikely that hash collisions will happen during the parallel computation within one dimension. Hence, one notable advantage of this task structure is that, if different dimensions are processed sequentially, there will be no thread contention for accessing the hash table. (Note, however, that this does not preclude parallel computation of dimensions.)

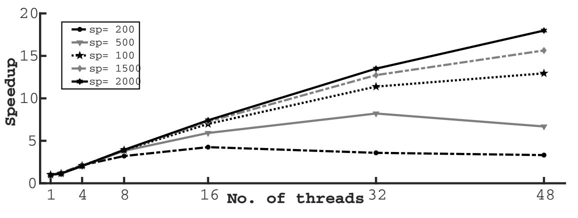

Fig. 8. Speedup versus the number of slices: speedup for the results presented in Fig. 7. As the number of slices increases, so does the efficiency gain from multi-core architectures. With $sp = 200$, the number of slices per core can vary from 200 to 4, depending upon the number of threads.

**5.2. Parallel task structure.** In this fine-grained 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. Algorithm 3 can be used as a parallel implementation of line 6 in Algorithm 1, or of line 11 in Algorithm 2.

**Algorithm 3.** Parallel computation of signatures for all dense units in a density chunk. $DU_i$ denotes the key of the $i$-th point in dense unit $DU$.

```
Require: $DC$: density chunk of keys in dimension $d$,
    $hTable$: common hash table (or hash table slice)
1: for each dense unit $DU \subseteq DC$ in parallel do
2:     $signature \leftarrow 0$
3:     for $i \leftarrow 0$ to $\tau$ do
4:         $signature \leftarrow signature + DU_i$
5:         Hash $signature$ to $hTable$
6:     end for
7: end for
```

As $\tau$ 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 (Nvidia CUDA, 2018). 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, for example, to determine memory locations for reading input and writing output data. In our case, an ID is required to identify the dense unit to process.

**5.3. Computing subsets efficiently.** Algorithm 3 presupposes that each task knows how to retrieve its dense unit $DU$, which is a subset $\{P_{i_0}, \ldots, P_{i_\tau}\}$ 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 advance to the lexicographically next subset from a given one (Loughry *et al.*, 2000).

In our parallel scenario, however, each thread needs to identify its relevant subset independently, without referring other results. More precisely, a thread with ID $i$ must find the $i$-th subset without accessing the $(i-1)$-th or any other previous subset. The parallel computation of all subsets of size $\tau+1$ of a given set is significantly more complex than advancing to the next subset from a given one. We outline two ways to solve this problem.

**5.3.1. Precomputing bit representations.** One solution for enabling data-parallel processing of dense units—used in our initial implementation—is a small sequential precomputation step populating an array with 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 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. While the precomputation of the array is sequential, it uses a very efficient implementation for computing the lexicographically next bit permutation (Anderson, 2018).

Precomputing the dense unit array of length $\sim 500000$ for a density chunk of size $t = 60$ and dense units of size $\tau + 1 = 4$ takes about 12 ms on an Intel Core i7-4720 HQ @2.6 GHz. Computing an array of $\sim 50$ million permutations ($t = 60, \tau + 1 = 6$) takes 1252 ms. A parallelization of this precomputation may be possible, similarly to the idea of parallel prefix (Harris *et al.*, 2007).

In order to calculate the signatures for a dense unit, each thread $i$ uses the $i$-th bit string in the precomputed array as a bit mask for choosing the appropriate keys from the density chunk and sums them up. One downside of this approach is that for large density chunks, both the number of bit strings and their length get large, requiring considerable extra space.

**5.3.2. Combinatorial decomposition.** The combinatorial representation (or *combinadics*) of a non-negative integer $i$ allows a direct computation of the $i$-th subset of size $k$ of a given set (McCaffrey, 2004). It exploits the fact that any integer $i$ can be represented as a unique sum of $k$ binomial coefficients. More precisely,

$$\forall i \geq 0, k \geq 1 \quad \exists 0 \leq n_1 < \cdots < n_k : \qquad i = \sum_{j=1}^{k} \binom{n_j}{j}.$$

For example, if $k = 4$, we have

$$\begin{aligned} i = 6 &= \binom{0}{1} + \binom{1}{2} + \binom{3}{3} + \binom{5}{4} \\ &= 0 + 0 + 1 + 5, \\ i = 20 &= \binom{0}{1} + \binom{2}{2} + \binom{4}{3} + \binom{6}{4} \\ &= 0 + 1 + 4 + 15, \\ i = 99 &= \binom{3}{1} + \binom{4}{2} + \binom{6}{3} + \binom{8}{4} \\ &= 3 + 6 + 20 + 70. \end{aligned}$$

The numbers in the upper parts of the binomial coefficients correspond to the elements contained in the $i$-th subset of size $k$ (in lexicographical order). Using the above examples as an illustration, the 6th subset of size $4$ of the set of natural numbers would be $\{0, 1, 3, 5\}$, while the 20th subset would be $\{0, 2, 4, 6\}$ and the 99th subset would be $\{3, 4, 6, 8\}$.

A simple greedy algorithm can be used to compute the combinatorial decomposition of $i$, and hence the $i$-th subset (McCaffrey, 2004). From a complexity point of view, this involves $O(\tau \times \log t)$ calculations of binomial coefficients, each with cost $O(\tau)$. Since the same binomial coefficients are computed repetitively, it makes sense to trade off some extra memory, providing a table of pre-calculated binomial coefficients in order to improve efficiency. For typical sizes of $t$ and $\tau$, the table of required binomial coefficients can be pre-calculated within some microseconds and stored in the GPU's read-only cache (also known as *constant memory*) for fast access by all threads. If for some large density chunk the table might not be sufficient, the relevant coefficients can always be calculated on demand.

**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 (Alcantara, 2011).

Our approach, based on the work by Strohm *et al.* (2015), is currently being implemented, and hence is not part of the evaluation presented here. Note that the GPU memory is a limiting factor for the hash table size. State-of-the-art GPUs are configured with up to 16 GB of RAM, which is sufficient to accommodate each partial table of the slicing approach described in Section 4.2.

**5.5. Performance evaluation.** Our current implementation of the GPU approach is a first step. It has not been optimized regarding the CUDA memory hierarchy, and hence it does not benefit from caching effects. Also, currently it does not use more intricate CUDA functions, 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.

They are shown in Table 1. The results do not include the time spent on pre-computation of subsets and on the hashing of signatures, but they cover all transfer times between GPU and host memory required for the GPU computations. For smaller numbers of signatures, GPU is slower than CPU. This was to be expected as there is

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

| #Signatures | CPU (ms) | GPU (ms) | Speedup |
|---|---|---|---|
| 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 |

always a small but non-negligible ramp-up cost for GPU kernels. Note that the speedup factor increases with the amount of calculations.

Also, the GPU used for this preliminary evaluation is a relatively smaller model; high-performance Tesla GPUs contain thousands of CUDA cores and achieve significantly higher processing power.

## 6. Conclusion

In this paper, we presented two independent approaches of parallelizing the SUBSCALE algorithm. First, we demonstrated the use of a shared memory multi-core architecture for parallelization. The CPU cores were assigned to the slices of the hash table computation and the results showed linear speedup with the number of cores.

The second approach uses fine-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 hash table slices) and many-core data parallelism for finer-grained subtasks (such as individual signature computation).

Future work includes combination of both approaches, making the best possible use of the different processing resources.

## References

Aggarwal, C.C. and Reddy, C.K. (2013). *Data Clustering: Algorithms and Applications*, 1st Edn., Chapman & Hall/CRC.

Aggarwal, C.C., Wolf, J.L., Yu, P.S., Procopiuc, C. and Park, J.S. (1999). Fast algorithms for projected clustering, *SIGMOD Record* **28**(2): 61–72.

Agrawal, R., Gehrke, J., Gunopulos, D. and Raghavan, P. (1998). Automatic subspace clustering of high dimensional data for data mining applications, *ACM SIGMOD International Conference on Management of Data, Seattle, WA, USA*, Vol. 27, pp. 94–105.

Alcantara, D.A.F. (2011). *Efficient Hash Tables on the GPU*, PhD thesis, University of California Davis, Davis, CA.

Anderson, S.E. (2018). Bit Twiddling Hacks–compute the lexicographically next bit permutation, `http://graphics.stanford.edu/~seander/bithacks.html#NextBitPermutation`.

Berkhin, P. (2006). A survey of clustering data mining techniques, *in* J. Kogan *et al.* (Eds.), *Grouping Multidimensional Data*, Springer, Berlin/Heidelberg, pp. 25–71.

Cheng, C.-H., Fu, A.W. and Zhang, Y. (1999). Entropy-based subspace clustering for mining numerical data, *5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA*, pp. 84–93.

Dagum, L. and Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming, *IEEE Computational Science Engineering* **5**(1): 46–55.

Datta, A., Kaur, A., Lauer, T. and Chabbouh, S. (2017). Parallel subspace clustering using multi-core and many-core architectures, *in* M. Kirikova *et al.* (Eds.), *New Trends in Databases and Information Systems*, Springer International Publishing, Cham, pp. 213–223.

Elhamifar, E. and Vidal, R. (2013). Sparse subspace clustering: Algorithm, theory, and applications, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **35**(11): 2765–2781.

Ester, M., Kriegel, H.-P., Sander, J. and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise, *International Conference on Knowledge Discovery and Data Mining, Portland, OR, USA*, pp. 226–231.

Fan, J., Han, F. and Liu, H. (2014). Challenges of big data analysis, *National Science Review* **1**(2): 293–314.

Fukunaga, K. (1990). *Introduction to Statistical Pattern Recognition*, Academic Press, San Diego, CA.

Geiger, A., Lenz, P., Stiller, C. and Urtasun, R. (2013). Vision meets robotics: The KITTI dataset, *The International Journal of Robotics Research* **32**(11): 1231–1237.

*Google Scholar* (2018). Search for 'data clustering', `https://scholar.google.com/scholar?q=data+clustering&btnG=`.

Han, J., Kamber, M. and Pei, J. (2011). *Data Mining: Concepts and Techniques*, 3rd Edn., Morgan Kaufmann Publishers, San Francisco, CA.

Harris, M., Sengupta, S. and Owens, J.D. (2007). Parallel prefix sum (scan) with CUDA, *GPU Gems* **3**(39): 851–876.

Jain, A.K. and Dubes, R.C. (1988). *Algorithms for Clustering Data*, Prentice-Hall, Inc., Upper Saddle River, NJ.

Jain, A.K., Murty, M.N. and Flynn, P.J. (1999). Data clustering: A review, *ACM Computing Surveys* **31**(3): 264–323.

Joliffe, I.T. (2002). *Principle Component Analysis*, 2nd Edn., Springer, New York, NY.

Jun, J., Chung, S. and McLeod, D. (2006). Subspace clustering of microarray data based on domain transformation, *VLDB Workshop on Data Mining and Bioinformatics, Seoul, Korea*, pp. 14–28.

Kailing, K., Kriegel, H.-P. and Kröger, P. (2004). Density-connected subspace clustering for high-dimensional data, *SIAM International Conference on Data Mining, Lake Buena Vista, FL, USA*, Vol. 4, pp. 246–256.

Kaur, A. and Datta, A. (2014). Subscale: Fast and scalable subspace clustering for high dimensional data, *IEEE International Conference on Data Mining Workshop, Shenzhen, China*, pp. 621–628.

Kaur, A. and Datta, A. (2015). A novel algorithm for fast and scalable subspace clustering of high-dimensional data, *Journal of Big Data* **2**(1): 1–24.

Kriegel, H.-P., Kröger, P. and Zimek, A. (2009). Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering, *ACM Transactions on Knowledge Discovery from Data* **3**(1): 1–58.

Li, T., Ma, S. and Ogihara, M. (2004). Document clustering via adaptive subspace iteration, *27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Sheffield, UK*, pp. 218–225.

Lichman, M. (2013). UCI machine learning repository, `http://archive.ics.uci.edu/ml`.

Loughry, J., van Hemert, J. and Schoofs, L. (2000). Efficiently enumerating the subsets of a set, `http://www.applied-math.org/subset.pdf`.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations, *5th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, CA, USA*, Vol. 1, pp. 281–297.

McCaffrey, J. (2004). Generating the MTH lexicographical element of a mathematical combination, *MSDN Library*, Microsoft, Redmond, WA.

Murtagh, F. (1983). A survey of recent advances in hierarchical clustering algorithms, *The Computer Journal* **26**(4): 354–359.

Nagesh, H., Goil, S. and Choudhary, A. (2001). Adaptive grids for clustering massive data sets, *1st SIAM International Conference on Data Mining, Chicago, IL, USA*, pp. 1–17.

Nvidia CUDA (2018). CUDA parallel computing platform and programming model, `http://www.nvidia.com/object/cuda_home_new.html`.

Parsons, L., Haque, E. and Liu, H. (2004). Subspace clustering for high dimensional data: A review, *ACM SIGKDD Explorations Newsletter* **6**(1): 90–105.

Sim, K., Gopalkrishnan, V., Zimek, A. and Cong, G. (2013). A survey on enhanced subspace clustering, *Data Mining and Knowledge Discovery* **26**(2): 332–397.

Steinbach, M., Ertöz, L. and Kumar, V. (2004). The challenges of clustering high dimensional data, *in* L.T. Wille (Ed.), *New Directions in Statistical Physics*, Springer, Berlin/Heidelberg, pp. 273–309.

Strohm, P.T., Wittmer, S., Haberstroh, A. and Lauer, T. (2015). GPU-accelerated quantification filters for analytical queries in multidimensional databases, *in* N. Bassiliades *et al.* (Eds.), *New Trends in Databases and Information Systems II*, Springer, Cham, pp. 229–242.

Thalamuthu, A., Mukhopadhyay, I., Zheng, X. and Tseng, G.C. (2006). Evaluation and comparison of gene clustering methods in microarray analysis, *Bioinformatics* **22**(19): 2405–2412.

Tierney, S., Gao, J. and Guo, Y. (2014). Subspace clustering for sequential data, *IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA*, pp. 1019–1026.

Xu, D. and Tian, Y. (2015). A comprehensive survey of clustering algorithms, *Annals of Data Science* **2**(2): 165–193.

Xu, R. and Wunsch, D. (2005). Survey of clustering algorithms, *IEEE Transactions on Neural Networks* **16**(3): 645–678.

Zhu, B., Mara, A. and Mozo, A. (2015). CLUS: Parallel subspace clustering algorithm on spark, *in* T. Morzy *et al.* (Eds.), *New Trends in Databases and Information Systems*, Communications in Computer and Information Science, Vol. 539, Springer International Publishing, Cham, pp. 175–185.

Zhu, J., Liao, S., Lei, Z., Yi, D. and Li, S.Z. (2013). Pedestrian attribute classification in surveillance: Database and evaluation, *ICCV Workshop on Large-Scale Video Search and Mining (LSVSM'13), Sydney, Australia*, pp. 331–338.

**Amitava Datta** is a professor at the University of Western Australia in the School of Computer Science and Software Engineering. His research interests are in data mining, parallel and distributed computing, mobile and wireless computing, bioinformatics, social networks and software testing.

**Amardeep Kaur** has received her PhD degree in data mining from the University of Western Australia under Australia Endeavor Awards. Her research interests include data mining, parallel computing and information security.

**Tobias Lauer** is a professor of computer science in the Department of Electrical Engineering and Information Technology at the Offenburg University of Applied Sciences in Germany. His research interests include parallel computing on GPUs, data mining, business intelligence, and multidimensional databases.

**Sami Chabbouh** obtained his BEng degree in electrical engineering and information technology within a tri-national program jointly offered by the Offenburg University of Applied Sciences (Germany), IUT Haguenau (France), and Haute Ecole Arc Neuchâtel (Switzerland). His thesis work included implementation and testing of GPU algorithms for subspace clustering. He is currently a freelance software developer.

Received: 10 February 2018
Revised: 29 May 2018
Re-revised: 6 June 2018
Accepted: 16 September 2018