Distributed mining of convoys in large scale datasets

Orakzai, Faisal; Pedersen, Torben Bach; Calders, Toon

Published in:
Geoinformatica

DOI (link to publication from Publisher):
10.1007/s10707-020-00431-w

Creative Commons License
Unspecified

Publication date:
2021

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Orakzai, F., Pedersen, T. B., & Calders, T. (2021). Distributed mining of convoys in large scale datasets. *Geoinformatica*, *25*(2), 353-396. https://doi.org/10.1007/s10707-020-00431-w

Noname manuscript No.
(will be inserted by the editor)

Distributed Mining of Convoys in Large Scale Datasets

Received: date / Accepted: date

Abstract Tremendous increase in the use of the mobile devices equipped with the GPS and other location sensors has resulted in the generation of a huge amount of movement data. In recent years, mining this data to understand the collective mobility behavior of humans, animals and other objects has become popular. Numerous mobility patterns, or their mining algorithms have been proposed, each representing a specific movement behavior. Convoy pattern is one such pattern which can be used to find groups of people moving together in public transport or to prevent traffic jams. A convoy is a set of at least m objects moving together for at least k consecutive time stamps where m and k are user-defined parameters. Existing algorithms for detecting convoy patterns do not scale to real-life dataset sizes. Therefore in this paper, we propose a generic distributed convoy pattern mining algorithm called DCM and show how such an algorithm can be implemented using the MapReduce framework. We present a cost model for DCM and a detailed theoretical analysis backed by experimental results. We show the effect of partition size on the performance of DCM. The results from our experiments on different data-sets and hardware setups, show that our distributed algorithm is scalable in terms of data size and number of nodes, and more efficient than any existing sequential as well as distributed convoy pattern mining algorithm, showing speed-ups of up to 16 times over SPARE, the state of the art distributed co-movement pattern

Faisal Orakzai
Université Libre de Bruxelles & Aalborg University
E-mail: ofaisal@ulb.ac.be & fmo@cs.aau.dk

Torben Bach Pedersen
Aalborg University
E-mail: tbp@cs.aau.dk

Toon Calders
University of Antwerp
E-mail: Toon.Calders@uantwerpen.be

mining framework. DCM is thus able to process large datasets which SPARE is unable to.

Keywords Spatio-temporal data mining · pattern · distributed · big data · MapReduce · cluster

1 Introduction

The increase in location data being generated by GPS equipped devices in the last couple of years has attracted researchers towards the analysis of such data for extraction of collective mobility behaviour. One pattern studied in this context is the *Convoy* Pattern [14] which is useful in many application domains. It can be used to find groups of people traveling together by public transport or to determine potential candidates for carpooling etc. A convoy is a group of at least m objects moving together for at least k time instants. These groups of objects are found by performing density based clustering such as DBSCAN [8] on object locations at each time instant followed by combining these clusters over the time dimension into convoys. For mining convoy patterns, various sequential algorithms [1, 13, 14, 25] have been proposed, however the existing sequential algorithms have been tested only on small datasets which can easily fit into memory and thus completely ignore data access optimizations.

The movement behaviour captured by variants of the convoy pattern, e.g., VCoDA proposed in [25] and evolving convoys proposed in [1], is context dependent, e.g., [14] allows convoy objects to have a density connection to each other through the objects that are not part of the convoy but [25] does not. However, none of the algorithms proposed for mining convoy pattern is scalable enough to tackle large mobility datasets. For instance, the most efficient convoy pattern mining algorithm out of the algorithms proposed in [14] took 100 seconds during an experimental run on a small dataset containing a couple of hours of movement of only 13 cattle. The huge sizes of current mobility datasets and the limitations of existing sequential algorithms demand the development of a parallel algorithm that can run on a set of loosely connected machines (cluster) and can produce results faster.

The state of the art in convoy mining is the work in [9]. The authors propose a generic framework *GCMP* for mining co-movement patterns and its implementation called the Star Partitioning and ApRiori Enumerator (SPARE) framework. The experiments show huge performance gains over sequential algorithms. SPARE considers the clustering phase of convoy mining as a preprocessing phase and only on the second phase of the pipeline which involves matching of the clusters to discover convoys. This approach doesn't yield much benefits because the part that is optimized is dominated by the clustering phase, which is not optimized. Our algorithm provides an end-to-end optimized solution for convoy mining from data partitioning to convoy discovery which therefore outperforms SPARE by a great margin.

In this paper, we formalize the problem of mining convoy patterns in a distributed shared-nothing architecture and propose a generic *Distributed Convoy*

Mining algorithm *DCM*. We implemented our proposed algorithm using the Hadoop MapReduce framework, the open source implementation of Google's MapReduce [6] framework and the most widely used framework for parallel processing of huge data sets. Experiments show that our algorithm is linearly scalable in terms of number of machines and data size, and up to 16 times faster than the state of the art distributed co-movement pattern mining algorithm SPARE. DCM$_{MR}$ is able to process large datasets which SPARE is unable to.

In our proposed algorithm, we use a divide and conquer strategy and distribute the workload to different machines (nodes) in the cluster. The sequential algorithm consists of two major operations, density-based clustering and cluster matching. As density based clustering is the most expensive part of convoy pattern mining, we focus on balancing its load among the various nodes and are able to achieve ideal parallelism. However for cluster matching, parallelism varies depending on the type of data and number of partitions. We have seen the overall proportion of sequential execution time varying from 5% to 15%. We partition the data on its temporal dimension. In each node, our local convoy pattern mining algorithm DCM$_{part}$ runs DBSCAN on a partition and mines local convoys. The nodes then send the local convoys to a central node where our merging algorithm DCM$_{merge}$ combines them to produce a global result. Our experiments show that the merging process is not expensive and DCM achieves linear speedup with the number of nodes.

DCM is a shared-nothing distributed algorithm which can be run not only on a cluster of machines but also on multi-core machines. On multi-core machines, multiple (Java) processes (mappers/reducers in MapReduce case) are run in parallel, each having its own memory space and the ability to access disks in parallel. Thus DCM is also a multi-core algorithm which does not depend on shared memory like multi-threaded applications. Machines based on the Non-Unified Memory Access (NUMA) architecture are examples of scaling-up in which multiple cores are added to a single machine to achieve parallelism. In this paper, we report DCM's performance results on a multi-node cluster as well as a NUMA based machine.

This journal paper significantly extends our previous conference paper [19] which we

- formalize the distributed convoy pattern mining problem and propose a novel generic distributed convoy pattern mining algorithm, DCM, that is based on a divide and conquer strategy and is independent of any data processing framework.
- present an implementation of DCM using the Hadoop MapReduce framework which is called DCM$_{MR}$.

The extensions are as follows:

- We present a detailed theoretical analysis and cost model of DCM (section 8) and conduct experiments to validate our findings.
  - We model the effect of setting the number of partitions and show its effect in relation to the data distribution and provide guidelines for

  choosing the partition size (section 8.2). We show through experiments that a careless choice of partition sizes could slow down the algorithm significantly.
  - We model the effect of cluster utilization on DCM's performance (section 8.3).
- We validate our findings on different scaling strategies i.e. single machine, scale-up (multi-NUMA node vertical scaling) and scale out (multiple machines) whereas previous work only considers multi-machine scale with Uniform Memory Access architecture (UMA) (appendix 9).
- We conduct extensive experimentation on much larger and more diverse datasets using different user parameters on 3 different scaling strategies. We compare DCM with the state of the art SPARE framework and show that DCM is up to 16 times faster than SPARE on the Trucks and Brinkhoff datasets. DCM successfully processes the TDrive dataset which SPARE fails to do. (section 9.1-9.5).
- We conduct experiments to analyse the behaviour of DCM depending on different factors that have an impact on DCM's performance e.g. partitioning size, false convoys etc. (section 9.6).

The paper is structured as follows:

- Section 2 covers the related work.
- Section 3 formalizes the convoy mining problem.
- Section 4 explains the choice of data partitioning strategy.
- Sections 5 and 6 explain the DCM algorithm.
- Section 7 explains our MapReduce implementation of the DCM algorithm.
- Section 8 covers the theoretical analysis of DCM.
- Section 9 contains the experimental results.
- Section 10 contains the conclusion and future work.
- Appendix A discusses the NUMA architecture.

2 Related Work

A number of mobility patterns and their detection algorithms have been proposed. One of the first such patterns is the *flock* pattern [10, 23]. A flock is a group of objects moving together for a time period t within a disk with user-specified radius r. Although flocks are a good way of identifying objects moving in groups, the disk constraint limits its ability to represent objects moving together either in a shape other than the disk or in a disk shape greater than the size specified by the user. The choice of an appropriate size for the disk itself is a challenging problem. [17] presents a scalable algorithm for flock pattern mining called *Bitmap Disk Filtering* algorithm.

To avoid the size and shape constraints required by the flock pattern, Jeung et al. have proposed the *convoy* pattern in [13, 14]. Unlike the *flock* pattern, in a *convoy* pattern the objects which are density-connected to each other are considered to be together. Two points are density-connected if the distance between them is less than a threshold r and at least one of the points has at least $minPts$ number of points with in a circle of radius r or if there is a chain of points between them such that the distance between the consecutive points is less than the threshold r and each point in the chain has at least a specified number of points within a circle of radius r. This relieves the *convoy* pattern of any restriction of size and shape.

Kalnis et al. [15] proposed the notion of *moving cluster* which is a sequence of spatial clusters with a certain percentage θ of similarity between the clusters of consecutive timestamps. A *moving cluster* maintains its identity while objects join or leave, whereas in a *convoy*, the objects should be together throughout the convoy's lifespan. Jung et al. proposed another variation of convoys in [1] called *evolving convoys* in which objects can join and leave during the lifespan. Tang et al. [22] proposed a buddy based approach for finding traveling companions from streaming trajectories. [18] offers an algorithm for finding loosely coupled groups of people which could also be from different clusters.

The primary reason for the poor performance of existing algorithms is the cost complexity of the original DBSCAN which is $\mathcal{O}(n^2)$ for each run where n is the number of objects present at the corresponding time-stamp. The DBSCAN algorithm issues a nearest neighborhood query for each point. Hence 3 hours of movement data of a million objects with a sampling frequency of 2 seconds, results in 540 million nearest neighborhood queries which may take days to process on a PC. The cost complexity of DBSCAN can be reduced to $\mathcal{O}(n \log n)$ by using an indexing structure but indexing huge amounts of data is expensive in terms of storage and maintenance. The memory cost of DBSCAN is $\mathcal{O}(n)$. To speed up DBSCAN, a distance matrix of size $(n^2 - n)/2$ can be constructed in memory but this increases the required memory to $\mathcal{O}(n^2)$. Existing algorithms, e.g., *CMC* [13] and *VCoDA* [25] are plagued with expensive clustering as clustering needs to be done for each timestamp thus they cannot scale to huge datasets. Parallel DBSCAN algorithms [5,11,21] are good for one large run but not suitable for many small DBSCAN runs.

The algorithms *CuTS, CuTS+* and *CuTS** [14] try to reduce the cost of DBSCAN by reducing the number of objects n in each run using a filter-and-refine paradigm. In the first phase, each object's trajectory is simplified by reducing the number of time-location pairs using the *Douglas-Peucker* algorithm (*DP*) [7]. The simplified trajectories are then partitioned into pieces each corresponding to a time duration λ. For each time duration λ, the pieces(sub-trajectories) are clustered using the DBSCAN algorithm. This step reduces the dataset to only those object trajectories which have the potential to form convoys. In the second step, the *CMC* algorithm is applied on the reduced dataset. Although this method has proven faster in previously discussed methods, the cost of trajectory simplification is $\mathcal{O}(T^2 N_t)$ where T is the number of points in a trajectory and N_t is the number of trajectories in the dataset. Using trajectory simplification also disallows us from using the same indexing structure as that of DBSCAN. The index for DBSCAN is based on location whereas the index required for trajectory simplification is based on object identity. The fact that the trajectory simplification process cannot use the existing spatial index constructed for DBSCAN, has completely been ignored as the implicit assumption is that the data fits in memory and disk seeks are not performed. This assumption renders these algorithms inappropriate for huge data sizes as our experiments demonstrate. Finding the right combination of parameters for *CuTs* family, for it to give acceptable performance is very hard and may involve multiple expensive iterations with no guarantee of success.

Yoon et al. [25], discovered that the convoy mining algorithms proposed by Jeung et al. [14] have serious problems with accuracy and recall. They refer to the convoy pattern proposed by Jeung et al. as *partially connected convoy* pattern and propose a contextually different version of the *convoy* pattern called *Valid Convoy*. They present a corrected version of *CMC* algorithm called *PCCD* (Partially Connected Convoy Discovery). As *CuTs* family of algorithms is based on *CMC*, they also have serious accuracy issues.

In [9], the authors propose a generic framework GCMP is proposed for mining co-movement patterns. The authors propose two parallel implementations of GCMP framework using Apache Spark as an underlying platform, a baseline implementation and a more optimal one called the Star Partitioning and ApRiori Enumerator (SPARE) framework. The experiments show huge performance gains over sequential algorithms. In both implementations, there are two stages of mapreduce jobs connected in a pipeline manner. The first stage deals with spatial clustering of objects in each timestamp (which they call a snapshot), which can be seen as a preprocessing step for the subsequent pattern mining phase. In particular, for the first stage, the timestamp is treated as the key in the map phase and objects within the same snapshot are clustered (DBSCAN or disk-based clustering) in the reduce phase. Finally, the reducers output clusters of objects in each snapshot, represented by a list of key-value pairs. The clusters are then provided to the second phase of the pipeline as input in which patterns are mined from the clusters. For convoy mining, the first phase is the most expensive one, however, the baseline implementation and SPARE both focus only on the second phase of the pipeline and consider the first phase as the a preprocessing phase. This approach leads to larger overall convoy mining execution times.

Our previous paper [20] presents a comparison of different data partitioning strategies for distributed convoy mining and [19] proposes a distributed/parallel convoy mining algorithm *DCM* which can process larger data sizes. The presented algorithm is generic and platform/framework independent. The performance achieved by the algorithm enables mining of convoy patterns in real world scenarios with huge data sizes. However, our experiments found that the performance of the algorithm varies significantly between different parameter settings for the same dataset. While operating on large datasets, for which the distributed algorithm has specifically been designed, one cannot afford to have multiple runs of the algorithm to find the parameters which give acceptable performance. There does not exist a detailed theoretical analysis of the algorithm that could support building performance models for optimal execution of the algorithm in real-world scenarios. Also, the existing work considers only UMA architectures with a focus on multi-machine scalability. The algorithm has not been validated for multi-core, multi-processor performance and in case of NUMA architectures, multi-node performance. This leaves out a big chunk of small, medium and large enterprises which either prefer multi-node NUMA based scale-out or use powerful multi-core/multi-processor NUMA machines for fulfilling their analytical needs. In this paper, we tackle both of the above mentioned problems. We present a detailed theoretical analysis of the algorithm backed by experimental results, that allows the user to make an informed decisions regarding the choice of partition sizes which we have identified as the single most influential parameter for the algorithm's performance. We also show its effects on the cluster utilization. We validate the performance of the algorithm on multi-core and multi-processor scale-out as well as multi-node scale-out on NUMA architectures. Additionally, we perform experiments on even larger and more diverse datasets as compared to the ones used in the previous work, including real-world and synthetics datasets, to test the algorithm's effectiveness in real-world scenarios. We additionally compare DCM with the state of the art SPARE framework to show the efficiency and performance gain achieved by DCM over SPARE.

3 Convoy Mining

Before we explain the convoy pattern, it is necessary to understand the concept of density connection [8]. Consider a point p in a set of points S and a distance threshold ϵ. The ϵ-neighborhood of point p can be defined as $B(p, \epsilon) = \{q \in S | d(p, q) \leq \epsilon\}$ where $d(p, q)$ represents the distance between two points. Given a point p, a distance threshold ϵ and an integer m, a point q is said to be *directly density reachable* from p if $q \in B(p, \epsilon)$ and $|B(p, \epsilon)| \geq m$. If there exists a chain of points $p_1, p_2, \ldots, p_n$ such that p_{i+1} is directly density reachable from p_i, p_n is said to be *density reachable* from p_1. Based on the previously defined terms, we can now define the notion of *density connected*.

Definition 1 ***(Density-Connected)*** *Given a set of points S, a point $p \in S$ is* density-connected *to a point $q \in S$ with respect to ϵ and m if there exists a point $x \in S$ such that both p and q are density-reachable from x.*

Density-connection is a way of defining *togetherness* which is free of the size and shape constraints of the flock pattern.

A trajectory of length n can be represented as follows:

$$\{(x_1, y_1, t), (x_2, y_2, t + 1), ..., (x_n, y_n, t + n - 1)\}$$

Conceptually, a movement dataset consists of a set of trajectories representing the movement of different objects. Generally, an object can have many trajectories, each representing a particular trip but sometimes all the trajectories corresponding to an object are joined together such that each object has only one trajectory that represents its complete movement history. In this

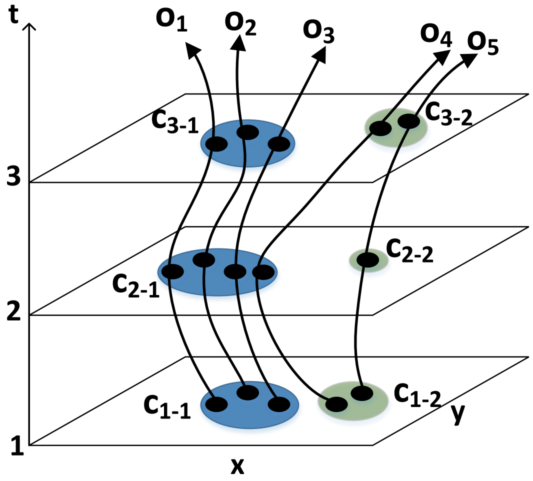

Fig. 1 Convoy Openness

paper, we do not assume any such abstraction and take movement data in its raw form as it arrives from location sensors.

At a physical level, movement data is stored in a 4-column table with schema $< oid, x, y, t >$ where *oid* is the object id, x and y represent the location in two dimensions, and t is the time instant at which the object was at that location. The trajectory of an object o can be extracted from this data by retrieving all the tuples with $oid = o$ and ordering them by time column t. A set of all density connected objects at a time instant is called a *snapshot cluster c*.

Definition 2 (Snapshot Cluster) *Given an integer m and distance threshold ξ, a set c of density connected points at time instant t w.r.t ξ is called a* snapshot cluster *if $|c| \geq m$, all $p \in c$ are mutually density connected and $\nexists r \notin c$ that is density reachable from any point in c.*

In Figure 1, for $m = 3$, clusters c_{1-1}, c_{2-1} and c_{3-1} are snapshot clusters as they are sets of density connected points with $size \geq m$. A *trajectory* is the movement trace of an object and contains all the points (x, y) traversed by the object in the order of traversal. Each point has a time t associated with it. In other words, a trajectory can be described as a temporal version of a line. A snapshot clusters at a time t_x can be found by querying all tuples with $t = t_x$ to retrieve all objects present at time t_x and their locations, and performing density based clustering. Jeung et al. define a convoy query as follows:

Definition 3 ***(Convoy Query)*** *Given a set of trajectories of N objects, a distance threshold ξ, an integer m, and an integer lifetime k, the convoy query returns all possible groups of objects, so that each group consists of a (maximal) set of density-connected objects with respect to ξ and m during at least k consecutive time points.*

Using the above definition, a convoy can be defined as:

Definition 4 ***Convoy*** *Given a set of objects O, a convoy v is a pair $(O, [t_s, t_e])$ such that $|O| \geq m$, $t_e - t_s + 1 \geq k$ and for all $t_s \leq t \leq t_e$, O is density connected at time t. Furthermore, v cannot be further extended, that is: for all $x \notin O, (O \cup \{x\}, [t_s, t_e]), (O, [t_s - 1, t_e]), (O, [t_s, t_e + 1])$ do not satisfy this constraint.*

In Figure 1, the set of objects $O = \{o_1, o_2, o_3\}$ forms a convoy with $m = 3$ and $k = 3$ which can be represented as $(O, [1, 3])$. To define a convoy in terms of snapshot clusters, we can say that it is a set of at least m objects present in the same snapshot cluster for k consecutive timestamps. A single snapshot cluster can be considered as a convoy with $k = 1$.

Table 1 Summary of Notation

Symbol	**Meaning**
v	A convoy
V	The set of all convoys
$V(P)$	The set of convoys in partition P
$t_s(x)$	Start time of a convoy or a partition
$t_e(x)$	End time of a convoy or a partition
c_{t-i}	i^{th} snapshot cluster at time instant t
C_t	A set of all snapshot clusters at time t
DB	Whole dataset
T_{DB}	Time interval of DB i.e. $[t_s(DB), t_e(DB)]$
P_i	i^{th} partition of DB
T_i	Time interval of i^{th} partition i.e. $[t_s(P_i), t_e(P_i)]$
N	Number of partitions
$V_C(P)$	The set of closed convoys in partition P
$V_L(P)$	The set of left-open convoys in partition P
v_L	A left-open convoy P
$V_R(P)$	The set of right-open convoys in partition P
$V_{LR}(P)$	A set of left-right-open convoys in partition P
$V_{part}(P)$	Output of DCM$_{part}$ from partition P
V_{global}	Global convoy result
$V^{x,y}$	A set of convoys with $m = x$ and $k = y$

4 Partitioning Strategies

In [20], we analyze different partitioning strategies for convoy pattern mining. Table 2 shows the comparison of the partitioning strategies with respect to the different properties which impact the performance of distributed algorithms. As it can be seen, temporal partitioning strategy is more suitable to distributed convoy mining.

Let DB be the complete set of movement data, N be the number of computational nodes available. Let $t_s(DB) = t_0 < t_1 < t_2 \cdots < t_{N-1} < t_N = t_e(DB)$. The time coherent partitioning of DB based on the split points

Table 2 Partitioning Strategies Comparison

Properties	**Object**	**Spatial**	**Temporal**
Data Exchange	Too High	Too High	Low
Data Redundancy	variable	High	No
Partitioning Cost	O(n)	O(n)	No
Disk Seeks	High	High	No
Data Ordering	No	No	Yes

$t_0, \dots, t_N$ is defined as: $DB = P_1 \cup \dots \cup P_N$ where: $P_i = \{(oid, x, y, t) \in DB | t_{i-1} < t \leq t_i\}$. Here, $t_s(P_i)$ and $t_e(P_i)$ denote the start and end time of the partition P_i. Notice that:

$$t_s(P_i) = t_{i-1} + 1 \qquad t_e(P_i) = t_i$$
$$t_s(P_1) = t_s(DB) \qquad t_e(P_N) = t_e(DB)$$

The timespan T_i of partition P_i for $i = 1 \dots N$ is defined as:

$$T_i = [t_s(P_i), t_e(P_i)]$$

As in an embarrassingly parallel job, the total execution time is determined by the execution time of the slowest node [24], data should be equally distributed in all partitions so that any one partition does not take significantly longer time to process and hence act as a bottleneck. For uniformly distributed data over time, partitions with equal number of time-stamps can be created with timespan:

$$T_i = \left[\left\lceil \frac{|T_{DB}|}{N} * (i-1) \right\rceil + 1, \left\lceil \frac{|T_{DB}|}{N} * i \right\rceil \right]$$

For the skewed datasets with imbalanced distribution of data over time, equal partitions over time is suboptimal. For such datasets, different partitioning strategies have been proposed in [4,12,16]. Some distributed data processing platforms implement their own partitioning strategy, e.g., MapReduce [6] partitions the data based on HDFS block size which ensures equal data size for each of its processing unit (Mapper).

As shown in table.2 temporal partitioning ensures no data exchange for local convoy pattern mining where as low data exchange in the global merging phase. As the data is temporally sorted naturally, there are no partitioning costs. The blocks in a distributed file system such as HDFS can be taken as partitions. Temporal partitioning also follows the co-location principle in both the phases of convoy pattern mining i.e. clustering (as all the locations for a timestamp exist on one node) and merging (as data is naturally sorted temporally and temporally adjacent clusters are required to be merged).

If we consider running DBSCAN on a partition only without considering the correctness of the result with respect to the whole dataset, the cost of DBSCAN on spatially partitioned data is lesser than the cost on a temporally partitioned data. However, the result of its execution on a temporally partitioned data will be correct as if the algorithm was run on a non-partitioned data. Therefore the overall cost of execution of DBSCAN on a spatially partitioned data is higher than on temporally partitioned data.

5 Local Convoy Mining DCM$_{\mathbf{part}}$

Convoy pattern mining on individual partitions using existing algorithms cannot produce an accurate global result unless information from neighboring partitions is provided. For instance, in another example, if a convoy spans three partitions $P_5 - P_7$ such that the parts of the convoy in partitions P_5 and P_7 have lifetime less than k, a convoy pattern mining algorithm will not be able to find its parts in partitions P_5 and P_7 and only discover part of the convoy in P_6. Similarly, if we are mining for convoys of length 6 and there exists a convoy which spans two partitions P_i and P_{i+1} such that in both the partitions the lifetime of the convoy is 3, the convoy will not be detected in the individual partitions.

In this section we propose the DCM$_{part}$ (Distributed Convoy Mining over Partitions) algorithm which takes care of the problems mentioned above that arise in a shared-nothing architecture. DCM$_{part}$ is an extension of the CMC (Coherent Moving Cluster) Algorithm [14] by Jeung et al. and the PCCD (Partially Connected Convoy Discovery) algorithm [25] by Yoon et al. which proposes some corrections to the former for accurate convoy discovery. It runs on individual partitions P and produces local candidate convoys V_{part} which are then merged with the local convoys produced by other instances of DCM$_{part}$ running on different partitions to produce the global convoys V_{global}. The set of local convoys is the union of four types of convoy sets:

$$V_{part}(P) = DCM_{part}(P) = V_C(P) \cup V_L(P) \cup V_R(P) \cup V_{LR}(P)$$

We define each of the local convoy sets in the following; using the notation: $V^{m,k}(P)$ representing a set of convoys in partition P with minimum number of objects m and minimum lifetime k.

5.1 Closed Convoys: V_C

V_C is the set of local convoys with life $>= k$ that have no chances of extending to other partitions hence they do not take part in the merge process and are directly added to the global convoy result V_{global}. Such convoys do not touch partition boundaries with the exception, if it is the dataset boundary. V_C can be written as:

$$V_C^{m,k}(P) = \left\{ v \in V(P) \middle| \begin{array}{c} (t_s(v) \neq t_s(P) \vee t_s(v) = t_s(DB)) \\ \wedge(t_e(v) \neq t_e(P) \vee t_e(v) = t_e(DB)) \end{array} \right\}$$

For cases where the timespan of a partition is less than or equal to $k+1$, V_C is an empty set, hence:

$$|V_C(P_i)| \begin{cases} = 0 & \text{if } |T_i| \leq k+1 \\ \geq 0 & \text{if } |T_i| \geq k+2 \end{cases}$$

This is because all convoys with life $\geq k$ will touch the partition boundary and may extend to the neighboring partition. Therefore such convoys are not considered as locally closed. Figure 2 shows closed convoys in blue color.

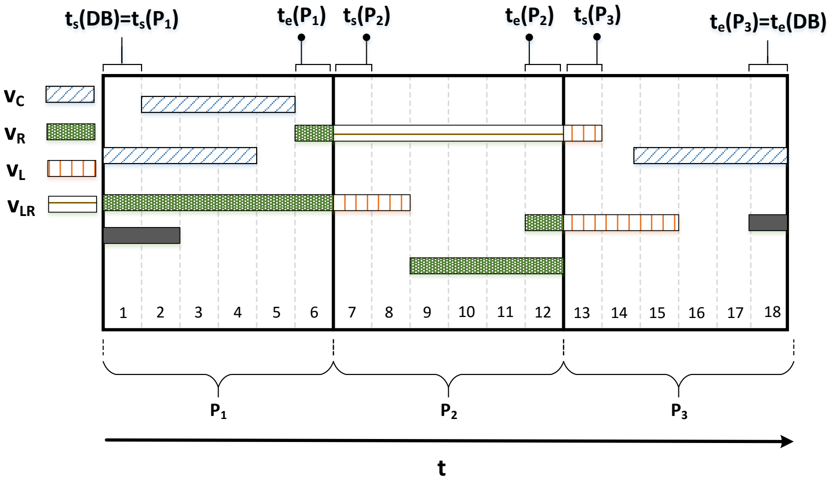

Fig. 2 Convoy Openness

5.2 Left-Open Convoys: V_L

V_L is the set of convoys that are closed from the right but have chances of getting extended to the left. While partitioning the data on the temporal domain, for ease of comprehension and visualization, we consider the time moving from left to right. A left-open convoy $v_L \in P_i$ lies at the left edge of partition P_i. The life-time of such convoys can be less than k but after merging them with right-open convoys from P_{i-1}, they may get extended and their extended life-time may satisfy k. In the first partition P_1 of DB, $V_L = \phi$ holds true as convoys cannot extend beyond the left edge of the first partition. The convoy set V_L can be written as follows:

$$V_L(P) = \left\{ v \in V^{m,1}(P) \left| \begin{array}{c} t_s(v) = t_s(P) \\ \wedge(t_e(v) \neq t_e(P) \vee t_e(v) = t_e(DB)) \\ \wedge t_s(P) \neq t_s(DB) \end{array} \right. \right\}$$

$V^{m,1}$ is a set of all convoys with parameter m as that of the convoy query but $k = 1$. A snapshot cluster c at time t is also a convoy with a lifetime of 1 and lifespan of $[t, t]$. Figure 2 shows left-open convoys in orange color.

5.3 Right-Open Convoys: V_R

The convoy set V_R contains all the convoys which are closed from the left but open from the right. Similar to $V_L(P)$, the set of right-open convoys is defined as:

$$V_R(P) = \left\{ v \in V^{m,1}(P) \,\middle|\, \begin{array}{c} t_e(v) = t_e(P) \\ \wedge(t_s(v) \neq t_s(P) \vee t_s(v) = t_s(DB)) \\ \wedge t_e(P) \neq t_e(DB) \end{array} \right\}$$

The green convoys in Figure 2 are right-open convoys.

5.4 Left-Right-Open Convoys: V_{LR}

The local convoy set V_{LR} contains all the convoys in the partition which are open at both ends. As discussed before, the first partition P_1 and the last partition P_N cannot have left-open and right-open convoys respectively, and implies $V_{LR}(P_1) = V_{LR}(P_N) = \phi$. The set V_{LR} can be written as:

$$V_{LR}(P) = \left\{ v \in V^{m,1}(P) \,\middle|\, \begin{array}{c} t_s(v) = t_s(P) \\ \wedge t_e(v) = t_e(P) \\ \wedge t_s(P) \neq t_s(DB) \\ \wedge t_e(P) \neq t_e(DB) \end{array} \right\}$$

The brown convoy in Figure 2 is a left-right-open convoy.

5.5 Convoy Types Disjointness

The types of partial convoys described above are mutually exclusive; i.e, a convoy cannot be left-open and right-open at the same time. However, we can identify each convoy with its type by using two relaxed openness properties which are the same as above properties but are not mutually exclusive; *right-open** and *left-open**. The * denotes relaxed properties. If a convoy is left-open* and *right-open** at the same time, it is a right-left-open convoy. Using relaxed properties for identification of the types makes the convoy mining algorithm simpler and more intuitive.

A relaxed local left-open* convoy set is defined as follows:

$$V_{L*}(P) = \left\{ v \in V^{m,1}(P) \,\middle|\, \begin{array}{c} t_s(v) = t_s(P) \\ \wedge t_s(P) \neq t_s(DB) \end{array} \right\} \quad (1)$$

Likewise, a relaxed right-open* convoy set is written as:

$$V_{R*}(P) = \left\{ v \in V^{m,1}(P) \,\middle|\, \begin{array}{c} t_e(v) = t_e(P) \\ \wedge t_e(P) \neq t_e(DB) \end{array} \right\} \tag{2}$$

It is easy to see that;

$$\begin{aligned} V_L(P) &= V_L^*(P) \setminus V_R^*(P) \\ V_R(P) &= V_R^*(P) \setminus V_L^*(P) \\ V_{LR}(P) &= V_L^*(P) \cap V_R^*(P) \end{aligned}$$

5.6 The Algorithm DCM$_{part}$

Algorithm 1 shows the pseudo-code of the DCM$_{part}$ algorithm. Note that multiple instances of DCM$_{part}$ can run on a node if it hosts multiple partitions. As a snapshot cluster is also a convoy with a lifetime of 1, we use identical storage structures for convoys and clusters. We use Figure 3 to illustrate different parts of the algorithm considering $m = 2$ and $k = 2$.

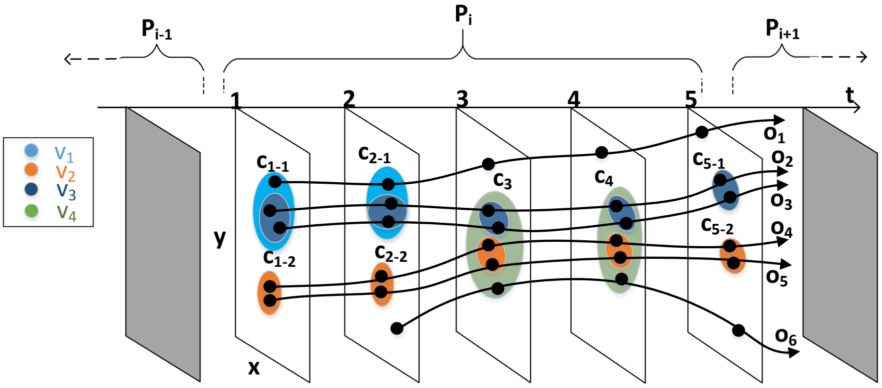

Fig. 3 Illustration for DCM$_{part}$

For each partition, DCM$_{part}$ reads the data starting from the start time of the partition. For each time-stamp, it retrieves all the objects and clusters them using DBSCAN algorithm (*line 3*). In Figure 3, for $t = 1$, the cluster set C contains clusters c_{1-1} (light blue) and c_{1-2} (orange). We initialize each cluster in the list of clusters C discovered by DBSCAN (lines 4-8) and mark them as *left-open** if they satisfy the conditions of (1) (lines 7-8).

Convoy set V contains all the candidate convoys discovered until time-stamp $t - 1$. We match all the convoys in V with all the clusters in C to find the convoys which extend to the current time t (lines 9-22). As in our example, we are still at the first time-stamp, the set V will be empty and the algorithm will continue at line 23. Each of the clusters c_{1-1} and c_{1-2} will be added to the convoy set V_{next} as candidate convoys. V_{next} is a set which contains all candidate convoys which have a potential of extending in the next iteration. V_{next} is taken forward to the next iteration as V. In the example, at the end of the first iteration, $V_{next} = \{v_1 = (o_{[1,3]}, [1,1]), v_2 = (o_{[4,5]}, [1,1])\}$. For brevity, we denote individual candidate convoys by $v^O_{[s,e]}$ where s, e and O are the start time, end time and the set of objects of the convoy respectively. In the next iteration, for $t = 2$, new clusters c_{2-1} (light blue) and c_{2-2} (orange) are found which are initialized and checked for left-open*.

The convoys in set V are called *candidate convoys* as they are not maximal yet. Each candidate convoy in V is matched with each newly discovered cluster in C by intersection. In our example, both $v^{1,2,3}_{[1,1]}$ and $v^{4,5}_{[1,1]}$ are elements of V at time $t = 2$ and are matched to c_{2-1} and c_{2-2}. At time $t-1$, if there is a convoy $v \in V$ such that $|v| \geq m$ and at time t, at-least m of its objects still form a cluster, the convoy is said to have *extended* (lines 12-13). In other words, a convoy v at time $t-1$ is said to have extended if the intersection between this convoy and any cluster at time t is greater than or equal to m.

In our example, the size of the intersection between $v^{1,2,3}_{[1,1]}$ and c_{2-1} is 3 which is greater than 2, hence the convoy gets extended and becomes $v^{1,2,3}_{[1,2]}$ (line 14). The end time of the extended convoy is updated to the current time $t = 2$. The left-open* property of a convoy is carried on to the extended convoy. The extended convoys are added to the set V_{next} to match them further in the next iteration. Addition to V_{next} is done through the *update* method (Algorithm 2) which ensures that all convoys in V_{next} are maximal.

If a convoy is a subset of the cluster we are matching it with, it is said to have been *absorbed* in the cluster (line 18). In Figure 3, at time $t = 3$, the orange convoy $v^{3,4}_{[1,2]}$ is a subset of the green cluster c_{3-2}, it is absorbed in the cluster. An absorbed convoy is a subset of its extended version; therefore it is discarded. A non-absorbed convoy acts as an independent convoy if it satisfies the lifetime constraint k.

In Figure 3 at time $t = 3$, the light blue convoy does not get absorbed in the green cluster c_3 and satisfies the length constraint k; hence it is added to the result V_{part}. If a non-absorbed convoy does not satisfy the length constraint k but is left-open*, it is still added to the result because it may extend to the partition P_{i-1} and satisfy k. Similarly, non-absorbed clusters are added to V_{next} as they may extend to the next time-stamp to form a convoy (lines 23-24). At time $t = 3$, the green cluster c_3 is an unabsorbed cluster as it is not a subset to any of the convoys (orange and light blue) from previous time-stamp $t = 2$. After the last iteration, the convoy set V contains all candidate convoys that may extend to the next partition P_{i+1}. They are checked for the right-open* property and are made part of the result V_{part} (lines 26-28).

6 Global Merge with DCM$_{merge}$

After local convoy pattern mining on each partition, the local results are sent to a single node for merging. For ease of description, we call it the *central*

Algorithm 1 DMC_{part}

```
Require: Partition $P$, Convoy size parameter $m$, Convoy lifetime $k$, Distance threshold $\epsilon$
Ensure: Local Convoy Set $V_{part}$
$V \leftarrow \phi$;$V_{part} \leftarrow \phi$
for each time $t \in [t_s(P), t_e(P)]$ (in ascending order) do
    $V_{next} \leftarrow \phi$; $O_t \leftarrow P(t)$ ; $C \leftarrow$ DBSCAN$(O_t, \epsilon, m)$
    for each cluster $c \in C$ do
        $c$.matched ← false; $c$.absorbed ← false;
        $c$.lifetime ← $[t, t]$
        if $t_s(c) = t_s(P)$ and $t_s(P) \neq t_s(DB)$ then
            $c$.leftOpen* ← true;
    for each candidate convoy $v \in V$ do
        $v$.extended ← false; $v$.absorbed ← false
        for each snapshot cluster $c \in C$ do
            if $|c \cap v| \geq m$ then
                $v$.extended ← true; $c$.matched ← true
                $v_{ext} \leftarrow c \cap v$
                $v_{ext}$.lifeTime ← [$v$.startTime , $t$]
                $v_{ext}$.leftOpen* ← $v$.leftOpen*
                $V_{next} \leftarrow$ update$(V_{next}, v_{ext})$
                if $v \subseteq c$ then $v$.absorbed ← true
                if $c \subseteq v$ then $c$.absorbed ← true
        if $v$.absorbed = false then
            if $v$.leftOpen* = true or $v$.lifetime $\geq k$ then
                $V_{part} \leftarrow V_{part} \cup \{v\}$
    for each $c \in C$ do
        if $c$.absorbed = false then $V_{next} \leftarrow V_{next} \cup \{c\}$
    $V \leftarrow V_{next}$;
for each candidate convoy $v \in V$ do
    if $t_e(P) \neq t_e(DB)$ then $v$.rightOpen* = true
    $V_{part} \leftarrow V_{part} \cup \{v\}$
return $V_{part}$
```

Algorithm 2 update

```
Require: Set of convoys $V$, Convoy to add $v_{new}$
Ensure: Updated convoy set $V$
for each candidate convoy $v \in V$ do
    if $v \subseteq v_{new}$ then $V \leftarrow V - v$
$V \leftarrow V \cup v_{new}$
return $V$
```

node. The local convoy set received at the central node can be written as:

$$V_{local} = \cup_{i=1}^{N} V_{part}(P_i)$$

The merging algorithm runs a customized matching process on candidate convoys to form true global convoys. The merge happens at the partition boundaries. We chose $t_e(P_i)$ as the partition boundary between P_i and P_{i+1} and call it *merge-time*. The set of all merge time-stamps MT is:

$$MT = \{t_e(P_1), t_e(P_2), \ldots t_e(P_{N-1})\}$$

Starting with the lowest merge-time, at each merge-time mt_i, all the right-open* convoys V_R* of partition P_i are matched with the left-open* convoys V_L* of the partition P_{i+1}. The merge-time of a convoy is given as follows:

$$mt(v) = \begin{Bmatrix} t_s(v) - 1 \text{ if } v \in V_L \vee v \in V_{LR} \\ t_e(v) \text{ if } v \in V_R \end{Bmatrix}$$

A left-right-open convoy can be merged to its right or left. As we perform the merging in an ascending order with respect to time a left-right-open convoy will first be merged at its left partition boundary and then at the right. After getting merged at the left, it gets closed from the left and becomes a right-open convoy. As the closed convoys do not have any mt, we use -1 as their mt.

Algorithm 3 shows the pseudo-code of the global merge algorithm. The algorithm takes as input a set of local convoys V_{local} and the convoy mining parameters m and k. Unlike the DCM_{part} algorithm in which we conduct the matching process for each time instant of the partition, we run the matching process for each mt. On line 2, the sets are initialized to be empty. V_{global} is the result set of the global merge algorithm, V_L* and V_R* represent the left-open* and right-open* convoy sets, respectively, that must be merged together in each iteration, and V_{Rnext*} denotes the set of right-open* convoys that we want to take to the next iteration of the matching process. After sorting, it becomes less costly to retrieve convoys with respect to their merge time in ascending order. As the set V_{local} also contains the closed convoys, they come on top of the set because we consider mt of closed convoys to be -1. Closed convoys do not need to go through the matching process so we extract them from local convoy set V_{local} and put them in the result set V_{global} (line 3). Corresponding to each mt in MT, we retrieve all convoys from V_{local}, group the right-open convoys into V_R* whereas the left-open and left-right-open convoys into V_L* and match them. Checking for the left-open* property first ensures that all left-open and left-right-open convoys land in to V_L* (lines 7-8).

Each convoy v_1 from V_R is matched with each convoy v_2 from V_L*. If their intersection is greater or equal to m, v_1 is considered as extended and a new extended convoy is formed whose start time is equal to the start time of the right-open* convoy v_1 and end time is equal to the end time of the left-open* convoy v_2 (lines 11-15). The extended convoy v_{ext} inherits the left-open* property from v_1 and the right-open* property from v_2 (lines 7-8). If in a merge operation, one of the convoys is a left-right-open convoy, the resulting convoy will always be a right-open convoy.

If the extended convoy v_{ext} is right-open*, we put it in $V_{Rnext}*$ so that we can take it to the next iteration and match it with left-open* convoys of the next partition boundary (line 19). If it is not right-open, it is a closed convoy and cannot be further extended. If its lifetime is greater than the minimum required lifetime k, we put it in the result set V_{global} otherwise we discard it (lines 20-22). Each time we update a convoy set, we use the function **update** to make sure that the convoy set contains only maximal convoys (it does not contain any sub-convoys). If the convoy v_1 is a subset of v_2, it is considered as absorbed which means that v_1 cannot act as an independent convoy but

Algorithm 3 DCM$_{merge}$

```
Require: Set of local convoys $V_{local}$, Partition boundaries $T_B$, Convoy size parameter $m$,
    Convoy lifetime $k$
Ensure: Global convoy set $V_{global}$
 1: sort($V_{local}$, $merge\text{-}time$)
 2: $V_{global} \leftarrow \phi$; $V_L* \leftarrow \phi$; $V_R* \leftarrow \phi$; $V_{Rnext}* \leftarrow \phi$
 3: $V_{global} \leftarrow V_{global} \cup V_{local}(-1)$
 4: for each partition boundary $t \in T_B$ do
 5:     $V \leftarrow V_{local}(t)$
 6:     for each convoy $v \in V$ do
 7:         if $v$.leftOpen* then $V_L* \leftarrow V_L* \cup \{v\}$
 8:         if $v$.rightOpen* then $V_R* \leftarrow V_R * \cup \{v\}$
 9:     for each convoy $v_1 \in V_R*$ do
10:         for each convoy $v_2 \in V_L*$ do
11:             if $v_1 \cap v_2 \geq m$ then
12:                 $v_1$.extended $\leftarrow$ true
13:                 $v_{ext} \leftarrow v_1 \cap v_2$
14:                 $v_{ext}$.startTime $\leftarrow$ $v_1$.startTime
15:                 $v_{ext}$.endTime $\leftarrow$ $v_2$.endTime
16:                 $v_{ext}$.leftOpen* $\leftarrow$ $v_1$.leftOpen*
17:                 $v_{ext}$.rightOpen* $\leftarrow$ $v_2$.rightOpen*
18:                 if $v_{ext}$.rightOpen* then
19:                     $V_{Rnext}* \leftarrow$ update($V_{Rnext}*, v_{ext}$)
20:                 else if $v_{ext}$.lifetime $\geq k$ then
21:                     $v_{ext}$.closed $\leftarrow$ true
22:                     $V_{global} \leftarrow$ update($V_{global}, v_{ext}$)
23:                 if $v_1 \subseteq v_2$ then $v_1$.absorbed $\leftarrow$ true
24:                 if $v_2 \subseteq v_1$ then $v_2$.absorbed $\leftarrow$ true
25:         if $v_1$.absorbed = false and $v_1$.lifetime $\geq k$ then
26:             $v_1$.closed $\leftarrow$ true
27:             $V_{global} \leftarrow$ update($V_{global}, v_1$)
28:     for each convoy $v_2 \in V_L*$ do
29:         if $v_2$.absorbed = false then
30:             if $v_2$.rightOpen* then
31:                 $v_2$.leftOpen* $\leftarrow$ false
32:                 $V_{Rnext}* \leftarrow$ update($V_{Rnext}*, v_2$)
33:             else if $v_2$.lifetime $\geq k$ then
34:                 $v_2$.closed $\leftarrow$ true
35:                 $V_{global} \leftarrow$ update($V_{global}, v_2$)
36:     $V_R* \leftarrow V_{Rnext}*$; $V_L* \leftarrow \phi$
37: return $V_{global}$
```

becomes part of the extended bigger convoy that results from the intersection between v_1 and v_2 (line 23). If the convoy v_1 does not get absorbed in the extended convoy and it fulfills the minimum lifetime condition k, we close it and put it in the result set V_{global} using the **update** method (line 25-27). After the end of each matching iteration, we check the status of each convoy v_2 part of V_L*.

If a convoy v_2 has not been absorbed into the extended convoy, it can act as an independent convoy. If it is right-open*, it is a left-right-open convoy so we put it in $V_{Rnext}*$ so that it can be further extended in the next iteration (lines 29-32). If it is not right-open*, we check if it satisfies the minimum lifetime condition for a convoy. If yes, we make it part of the result set V_{global} otherwise we discard it (lines 33-35). After this, we prepare the sets V_R* and V_L* for the next iteration. Note that no left-open* convoys are carried to the next iteration. This is because all absorbed or non-absorbed left-open* convoys become left-closed.

7 The Hadoop Implementation (DCM$_{MR}$)

In this section we present the MapReduce-based implementation DCM$_{MR}$ for distributed convoy pattern mining which is based on our generic DCM algorithm proposed in the previous section. We use the MapReduce framework that is part of Apache Hadoop. DCM$_{MR}$ consists of two phases, the Map and the Reduce phase. In the Map phase, DCM$_{MR}$ runs DCM$_{part}$; whereas in the reduce phase, it runs DCM$_{merge}$, the global merge algorithm. Each instance of the Mapper runs on a single partition which is determined by the block size of the input data. The MapReduce framework splits files based on size without considering the content of the file. As DCM$_{MR}$ requires time coherent partitioning in which the data corresponding to one time-stamp is not split between partitions, we implement a custom InputFormat and RecordReader to ensure it. In Hadoop MapReduce framework, an InputFormat describes how to read data from a file whereas a RecordReader does the actual ground work of reading the data and providing it to the Mappers. The parameters $m, k, e, t_s(DB)$ and $t_e(DB)$ are passed to the Mappers through MapReduce Job Configuration.

Our input dataset consists of a text file in which each line is a quadruple $< oid, x, y, t >$. This file is stored in HDFS (Hadoop Distributed File System). HDFS divides the file into blocks and distributes them all over the cluster. Each block is processed by a Mapper which runs DCM$_{part}$ on it. Hadoop reads each block using our custom InputFormat and RecordReader and calls the Map function for each line of the data block. The Map function collects the tuples for a each timestamp and performs density-based clustering. The resulting clusters are used to perform local convoy pattern mining as they become available. When the whole block has been read and local convoy mining has been performed, DCM$_{part}$ transmits the set of local convoys to the reducer.

DCM$_{part}$ transmits each convoy in the form of a $< key, value >$ pair in which the key is the merge time $mt(v)$ of the convoy being transmitted. We use merge time of a convoy as its key to use the sorting function of MapReduce instead of sorting convoys in the reducer. As local convoy mining on different partitions will produce convoys with different merge times, Hadoop will fire up different reducers for each group of convoys with the same merge time. We do not want this because DCM$_{MR}$ requires a single instance of DCM$_{merge}$ to run. To solve this problem, we implemented a custom partitioner to make sure that all convoys landed in a single reducer. In addition to that we also implemented a custom grouping comparator so that all local convoys are grouped together and are passed to DCM$_{merge}$ running in the reducer in a single call. The *value* is a tuple of the form $< lo^*(v), ro^*(v), t_s(v), t_e(v), O(v) >$ where $lo^*(v)$ and $ro^*(v)$ are left-open* and right-open* properties of the convoy v being transmitted. In the Reduce phase, DCM$_{merge}$ is executed as-is except for the sorting of convoys because the convoys it receives are already sorted by the Hadoop.

8 Theoretical Analysis of DCM$_{MR}$

In this section, we detail the cost model of DCM and model the effect of setting the number of partitions and cluster utilization on the performance of DCM$_{MR}$.

8.1 Cost Complexity

The cost λ of the algorithm can be broken down as:

$$\lambda = \lambda(DCM_{part}) + \lambda(S) + \lambda(DCM_{merge})$$

The temporal cost of DCM$_{part}$ on a single partition is the sum of cost of clustering and matching in that partition. The clustering cost of a partition is the sum of cost of clustering at each timestamp in that partition. Let n be the total number of points in the dataset and T be the number of total timestamps. Assuming balanced distribution over time, the number of points in a timestamp can be written as n/T. As the cost of optimized DBSCAN algorithm is $\mathcal{O}(n\log(n))$, the cost of clustering points $\lambda_t(Cl)$ corresponding to a single timestamp is $\mathcal{O}\left(\frac{n}{T}\log\frac{n}{T}\right)$.

If there are p partitions of data, the number of timestamps in a partition can be written as T/p. Thus the total cost of clustering in a partition is:

$$\lambda_P(Cl) = \mathcal{O}\left(\frac{n}{T}\log(\frac{n}{T}) * \frac{T}{p}\right) = \mathcal{O}\left(\frac{n}{p}\log(\frac{n}{T})\right)$$

Let n_c be the number of clusters detected at each timestamp. In the matching process, each cluster of timestamp t is matched to each cluster of timestamp $t+1$. Thus the cost of matching between clusters of two consecutive timestamps is n_c^2. We assume that after matching the clusters of two consecutive timestamps, the number of partial convoys satisfying m is also equal to n_c. Thus the cost of matching $\lambda_P(M)$ in a single partition can be written as $\mathcal{O}(n_c^2(T/p-1))$.

DCM$_{merge}$ operates on open convoys sorted by their merge-time. It is easy to note that the number of merge times is always $p-1$. As p is generally not a huge number, the domain of sorting is much smaller than the range and we can use bucket sort with the cost of $\mathcal{O}(n)$. Let's say, on average, a partition emits the same number of left-open* and right-open* convoys equal to o, the cost of sorting $\lambda(S)$(which can be done in parallel) can be written as $\mathcal{O}\left(2o(p-2)+2o\right) = \mathcal{O}\left(2o(p-1)\right)$. After sorting, DCM$_{merge}$ has to merge the open convoys at their merge-time. The cost of merging $\lambda(DCM_{merge})$ can be written as $\mathcal{O}(o^2(p-1))$. The total cost of DCM can be written as:

$$\mathcal{O}\left(\frac{n}{p}log(\frac{n}{T}) + n_c^2(\frac{T}{p}-1) + 2o(p-1) + o^2(p-1)\right)$$

We have seen from observation and experiments that $o \approx n_c$ and $T \gg p$. For simplicity assuming that $p-1 \approx p$ and $T/p-1 \approx T/p$, we can write the total cost as:

$$\lambda(DCM) = \mathcal{O}\left(\frac{nlog(n/T) + n_c^2 T}{p} + p(n_c^2 + 2n_c)\right) \tag{3}$$

The first term in the equation above is the cost of the parallel part of DCM i.e. clustering and matching, whereas the last term represents the sequential part. For a constant n and T when $p \to \infty$, the run time is dominated by the sequential part i.e. DCM$_{merge}$. As $n \gg p$ and $T \gg n_c$, the parallel part of the algorithm is far greater than the sequential part consuming approximately 90% of the total runtime. The fact that the merge phase represents a smaller part of the total execution time justifies why we do not further parallelize the merge phase. Parallelizing the merge phase further, involves the use of multiple merge processes, each merging the output of a subset of DCM_{part} processes and then using a single DCM_{merge} process to merge the output of the intermediate DCM_{merge} processes. Note that there is also the cost of starting multiple intermediate DCM_{merge} processes which is generally significant in the current big data processing systems most of which are based on YARN (Yet Another Resource Negotiator)[1].

8.2 Effect of Partition Size

By default, DCM$^*_{MR}$ uses a block size of 128 MBs as partition size, which is Hadoop's default HDFS Block size. The partition size can also be specified by the user by passing it as a parameter to DCM$^*_{MR}$. In the datasets we used for the experiments, one partition holds the data corresponding to multiple timestamps which is what DCM$^*_{MR}$ expects. We call the data corresponding to a single timestamp a *time-block* (*tb*). Depending on the dataset being used, it could be the case that the time-blocks are bigger than the partition size i.e. a partition is not large enough to hold a time-block. This happens if the number of objects per timestamp is too large to be contained in the default partition size or if the user chooses a smaller partition size for execution. The size of a time-block can vary through out a dataset depending upon the number of objects active at different timestamps.

Figure 4 shows the effect of partition size on the processing times and Table 3 shows the ownership of time-blocks and partition processing times. The boxes

[1] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

Table 3 Time-block ownership and partition execution times

Partition	Time-blocks	Execution Time
1	$tb(t_1), tb(t_2), tb(t_3)$	2+1+2
2	$tb(t_1), tb(t_2)$	2+1
3	$tb(t_1)$	2
4	$tb(t_1)$	2

Table 4 Time-Block Ownership for Figure 5

Time-Block	**Owned By**
$tb(t_1)$	P_1
$tb(t_2)$	P_4
$tb(t_3)$	P_6

in blue color represent the time-blocks corresponding to timestamps t_1, t_2 and t_3 respectively. The time in the bracket denotes the time needed to process the data. In the figure, the smallest time-block corresponds to t_2 which can be denoted as $tb(t_2)$. The orange boxes represent 4 partitions of different sizes. A time-block belongs to the partition in which it starts. Reducing the partition size from P_1 to P_3 reduces the processing time of the partition but choosing a partition size smaller than the size of P_3 does not reduce the processing time further as a time-block cannot be divided. Choosing a partition size smaller than P_3 won't change the processing time of the partition.

Figure 5 shows a worst case scenario where the partition size is smaller than the smallest time-block. There are 9 partitions and only 3 time-blocks. To process this data, 9 instances of DCM$_{part}$ / Hadoop Mappers will be launched, each responsible for processing a single partition. For DCM$^*_{MR}$ to produce accurate results, a time-block should not be divided between partitions. Recall that a time-block belongs to the partition where it appears the first time. Table 4 shows the ownership of time-blocks by partitions for Figure 5. The rest of the partitions do not own any time-block. Instances of DCM$_{part}$ simply parse the data corresponding to these partitions but do not find any time-block starting

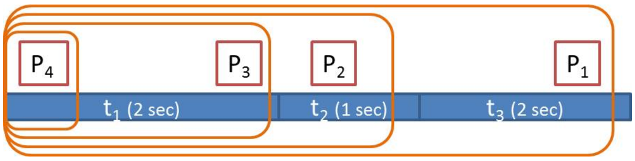

Fig. 4 Effect of Partition size

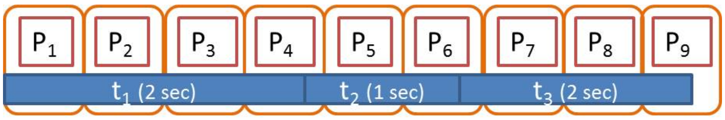

Fig. 5 Partition size smaller than minimum time data block

in the corresponding partition data. These DCM$_{part}$ instances only cause an IO penalty. Let $cost_{IO}$ be the penalty caused by a single non time-block owning DCM$_{part}$ instance, N^t be the total number of time-blocks, $cost(P_i)$ be the cost of processing a partition P_i, N^c be the number of cores, N^p be the total number of partitions, N^{p^*} be the time-block owning partitions and $N^{p'}$ be the total number of partitions not owning a time block, then the cost of Map phase execution can be written as:

$$cost(MapPhase) \cong \sum_{i=1}^{N^{p*}} cost(P_i^*) + cost(P') * N^{p'} \tag{4}$$

where $cost(P') = cost_{IO}(P')$. Note that $N^p = N^{p*} + N^{p'}$. Normally $N^{p'}$ is expected to be zero or negligible but when the number of partitions are increased, it tends to increase. In all cases, $N^{p*} \leq N^t$. When the specified partition size is less than the minimum time-block size, the number of time-block owning partitions becomes equal to the number of time-blocks. Hence:

$$N^{p*} \begin{cases} = N^t & \text{if } |P| \leq min(|tb(t_i)| \forall i \in [1, N^t]) \\ < N^t & \text{otherwise} \end{cases}$$

When N^p is increased, N^{p*} increases as well until it becomes constant. In the beginning the increase is sharp, whereas, in the end, just before becoming a constant, the increase is gentle. The variation in the increase of N^{p*} depends on the distribution of time-block sizes. The increase in N^{p*} means that the size of each time-block owning partition $|P^*|$ decreases, thus reducing the average Map processing time. The increase in $N^{p'}$ after N^{p*} becoming constant is directly proportional to the increase in N^p and drags the average Map execution time (MET) gently to $cost_{IO}$. Thus it can be written as:

$$\lim_{N^{p'} \to \infty} cost_{avg}(Map) = cost(P') = cost_{IO}$$

Figure 4 explains the effect of reduction in partition size over MET for time-block owning Mappers/instances of DCM$_{part}$. It can be seen in Table 3 that MET decreases by the decrease in partition size however when $|P| \leq |tb(t_1)|$, MET becomes constant which is the case for P_3 and P_4.

In case a partition size equivalent to P_1 is chosen, an instance of DCM$_{part}$ will process the whole data corresponding to t_1, t_2 and t_3 even though the last part of t_3 block does not belong to P_1. Similarly, DCM$_{part}$ responsible for processing P_2 will process the blocks of t_1 and t_2 even though the block of t_2 does not belong to P_2 completely. Thus the time to process partition P_2 will be 3 seconds. If we reduce the size of the partition further i.e. equivalent to P_3, the processing time will be 2 seconds. Notice that if we reduce the partition size further as in case of P_4, the processing time will not change as the instance of DCM$_{part}$ responsible for P_4 will process the complete t_1 block. Keeping in view the size of partition P_4, we can see that multiple partitions of same size can fit in the t_1 block thus t_1 block will be assigned to multiple partitions but only one instance of DCM$_{part}$ will process it. The rest of the instances will simply not process any data there as there is no time-block starting in the partitions allotted to them.

An increase in the number of splits also causes RET to increase linearly. This is because the number of intermediate open convoys increases as the number of time-block owning splits/partitions increase. In addition to causing extra network traffic, it leads to the Reducer processing more data. As the matching of the intermediate convoys is performed in a single Reducer, the total execution time of the algorithm also increases linearly with RET. This gives us an important insight about the choice of the partition size. Bigger the partition size, lesser the sequential part i.e higher the parallel part. But if the chosen partition size is increased such that:

$$\frac{|DB|}{|P|} \leq N^c$$

where $|DB|$ is the size of the complete dataset, the cluster will become under utilized as either one core will have less data to process or there will be less partitions to process than the number of cores available. Therefore the ideal choice of partition size is:

$$|P| = \frac{|DB|}{N^c}$$

8.3 Cluster Utilization

Another important aspect of DCM$^*_{MR}$ execution is its cluster utilization. Figure 6 shows the theoretical cluster utilization in the Map phase if Hadoop's default partitioning strategy (split size) is kept. The cluster utilization has been calculated as following:

$$util = \frac{N^p/N^c}{\lceil N^p/N^c \rceil}$$

If the number of partitions are larger than the number of cores available, the Mappers are run in phases. For example, if there are 6 partitions to process and 4 cores available, the processing will be done in two phases. In the first phase, 4 Mappers will be launched (1 Mapper per core) which will process the first 4 partitions. When any of the Mappers finishes and releases the occupied core, a new Mapper is launched to process one of the remaining partitions. Let us assume that it takes an equal amount of time to process a partition. In the first phase, 4 Mappers will finish at the same time, releasing all 4 cores. Now there are 4 cores available but only 2 partitions left. Therefore 2 Mappers will be launched utilizing only 2 out of 4 cores. Thus the cluster utilization in the second phase will be 0.5 or 50%. The overall cluster utilization in the Map phase according to the above formula will be: $\frac{6/4}{\lceil 6/4 \rceil} = \frac{1.5}{2} = 0.75$. This suggests that to have full theoretical cluster utilization, the number of partitions should be a multiple of the number of cores available i.e. $N^p \bmod N^c = 0$.

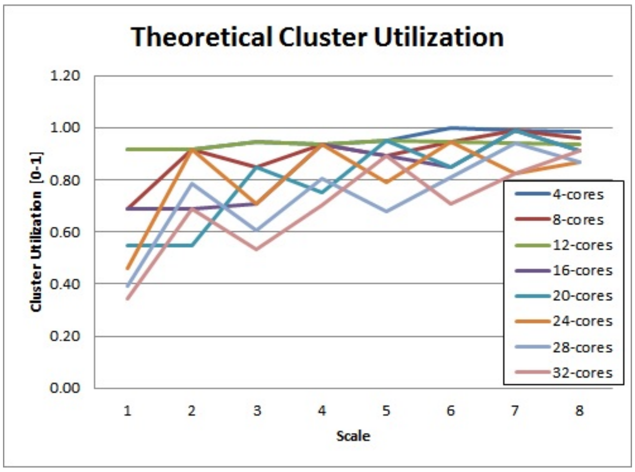

Fig. 6 Theoretical Cluster Utilization using Data Based Partitioning

8.4 Discussion on the Choice of Framework

DCM is a generic algorithm and does not depend on any framework. One can use Spark,MapReduce, Tez or Flink for its implementation. We used MapReduce as an implementation framework. DCM is an algorithm based on a divide and conquer strategy that maps naturally to the map-reduce framework with a relatively low number of map tasks and one reduce task. DCM is only one iteration of map-reduce so overhead of the framework is minimal; almost all time is spent on the algorithm itself, not on the framework, so the choice of the framework is right. The choice of framework, however, is not essential as the amount of overhead of the framework is minimal.

Apache Spark is a distributed data processing framework which performs better than Hadoop MapReduce framework on iterative jobs because of the following reason. Hadoop MapReduce persists the result of each Map and Reduce phase on disk for fault-tolerance purposes so that if an any iteration, the MapReduce job fails, it could restart from the result of the last iteration already persisted on disk. This helps in preventing the restart of the job from scratch. Spark however uses Resident Distributed Dataset (RDD) [28] which is an in-memory data structure that maintains the lineage information of all the operations applied to it and has the ability to recompute the lost part of the dataset in case of a fault/failure. DCM is not an iterative algorithm and reads the input data only once and writes the intermediate data to the disk which is only up to 3% more than the actual true result as shown in Figure 23. Another point to note here is that SPARE uses Spark and the experiments show that DCM is faster than SPARE even when using a framework that is considered slower than Spark.

9 Experimental Evaluation

Experiments were performed using the MapReduce implementation of our generic distributed convoy pattern mining algorithm, DCM$_{MR}$. We compared our algorithm with an optimized implementation of the single instance algorithm PCCD which we call PCCD* using a KD-Tree index for DBSCAN and the original implementation of the SPARE framework from its authors. For a fair comparison, we considered DCM$_{MR}$against SPARE for gain calculation, i.e. $Gain = \frac{T(SPARE)}{T(DCM_{MR})}$ where $T()$ represents the execution time of the algorithm. In terms of implementation, SPARE has the advantage of being implemented in Apache Spark which is an in-memory computation engine. DCM$_{MR}$has been implemented in the Hadoop MapReduce framework which uses disk to persist the intermediate results thus making it significantly slow compared to Apache Spark. Our experiments differ with the experiments in [9] as they consider the costlier clustering phase as a pre-processing step and do not include its execution time in the total execution time of the SPARE framework.

9.1 Infrastructure

For the experiments, three different hardware setups were used, each for single instance, scale-up (NUMA Architecture) and scale-out experiments. Following are the details of each setup.

9.1.1 Hardware Setup A (Single Instance)

For single machine multi-core experiments, a machine with a quad core Intel(R) Core(TM) i7-4700MQ processor with 2.4GHz frequency was used. The machine had 16GB RAM and a SCSI Solid State Drive. It ran a 64-bit Linux Mint 17.3 Rosa distribution with Linux 3.19.0-32-generic (x86 64) kernel. All the algorithms were run on Java(TM) SE Runtime Environment (build 1.8.0 101-b13) using the Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode) implementation. The figures for the experiments conducted on this setup are marked with "*(A)*".

9.1.2 Hardware Setup B (Scale-up)

For the scale-up set of experiments (NUMA based), we used a machine with 4 AMD Opteron(tm) 6376 processors running at 2.3 GHz. Each processor had 16 cores equally divided between 2 NUMA nodes. Thus the machine had 8 NUMA nodes, each node containing 8 cores making the machine's total core count to be 64. The machine had 512GB of RAM. It ran a 64-bit Ubuntu 14.04.3 LTS distribution with Linux 3.19.0-32-generic (x86 64) kernel. All the algorithms were run on Java(TM) SE Runtime Environment (build 1.8.0 101-b13) using the Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode) implementation. The figures for the experiments conducted on this setup are marked with "*(B)*".

9.1.3 Hardware Setup C (Scale-out)

We set up a cluster of 24 machines for the first set of experiments using a non-NUMA architecture. Each machine had 4GB RAM, a Core 2 Duo E8400 processor running at 3GHz and a 160GB SATA drive. It ran a 64-bit Ubuntu 14.04.3 LTS distribution with Linux 3.19.0-32-generic (x86 64) kernel. All the algorithms were run on Java(TM) SE Runtime Environment (build 1.8.0 101-b13) using the Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode) implementation. The figures for the experiments conducted on this setup are marked with "*(C)*".

9.2 Data Preparation and Parameter Setting

Following datasets were used for the experiments:

9.2.1 Trucks Dataset

The **trucks**[2] dataset consists of 276 trajectories of 50 trucks delivering concrete to several construction places around Athens metropolitan area in Greece. The locations in latitude and longitude were sampled approximately every 30 seconds for 33 days. To make the experiments compatible with the experiments performed on the trucks dataset in the previous papers [13, 14, 25], a single day's of a truck's movement was considered as a trajectory of a truck. The next day's trajectory of the same truck was considered as a different truck's trajectory to increase the number of objects in the dataset and hence, to find more convoys. For experiments comparing existing sequential algorithms, we used $m = 3$, $k = 180$ and $e = 0.0006$ as used in the original convoy paper [14].

9.2.2 T-Drive Taxi Dataset

[3,4] This dataset [26, 27] contains the GPS trajectories of 10,357 taxis during the period of Feb. 2 to Feb. 8, 2008 within Beijing. The total number of points in this dataset is about 15 million (29 million after interpolation) and the total distance of the trajectories reaches to 9 million kilometers. The average sampling interval is about 177 seconds with a distance of about 623 meters.

[2] http://chorochronos.datastories.org/

[3] http://research.microsoft.com/apps/pubs/?id=152883

[4] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/User_guide_T-drive.pdf

Table 5 Brinkhoff Dataset Properties

Property	Value	Property	Value
MaxTime	25000	data space height	26915
ObjBegin	5000	number of nodes	6105
ObjTime	100	number of edges	7035
ExtObjBegin	100	maximum time	25000
ExtObjTime	2	moving objects	2505000
data space width	23572	points	122014762

Table 6 Datasets used for data-scalability experiments

Objects (million)	Points (million)	Time Duration (min)
2	120	33.4
2.5	150	41.5
3	180	49.8
3.5	210	58.2
6.5	384	101.9

9.2.3 Brinkhoff Generator's Dataset

For performance comparison on synthetic datasets, we used the well-known Brinkhoff Generator [2,3] which can generate network based traffic data based on a real-world dataset and user specified parameters using simulation. It is open-source and publicly available. Table 5 shows different properties of the generated Brinkhoff dataset. The Brinkhoff generator does not support parameters large enough to generate millions of points as required by us for scalability tests. For instance, it does not accept to generate a dataset with more than 1000 time units or more than 5000 cars. Hence, we had to modify the source code to enable it to accept large parameters and generate meta data with each generated dataset. Table 6 shows the properties of the datasets used for the scalability experiments.

9.3 Results: Single Machine Experiments (Hardware Setup A)

To compare the performance of DCM$_{MR}$ with other algorithms on a single machine, we conducted experiments to see the behaviour of each algorithm with the increase in the number of cores allocated to the algorithms. The experiments were conducted on each of the datasets mentioned in section 9.2. Figure 7 shows that PCCD* being single threaded shows little change in the execution time with the increase in the number of cores. DCM$_{MR}$, although being a distributed algorithm is unable to show performance increase with the increasing number of cores because the size of the trucks dataset is small and most of the execution time is spent in the setup of the distributed job rather than the actual processing of the dataset. SPARE shows a sensitive behaviour towards the number of cores. Regardless of the size of the dataset, the complexity of the SPARE algorithm makes it resource hungry which results in performance improvement when more CPUs are provided. Overall, PCCD* performs the best on the smaller Trucks dataset.

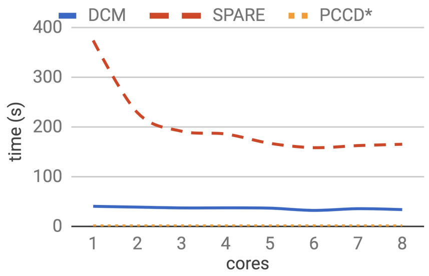

Fig. 7 (A) Execution times for Trucks dataset

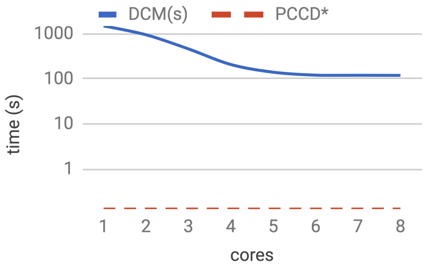

Fig. 8 (A) Execution times for TDrive dataset. SPARE did not finish in 3 hours

On the TDrive dataset (Figure 8), PCCD* performs best with up to 5 cores after which DCM$_{MR}$ surpasses it in performance. Even though the SPARE framework improves performance with more cores, shows higher execution times compared to DCM$_{MR}$ and PCCD* . PCCD* was unable to successfully process the Brinkhoff dataset (Figure 9) because of its big size. DCM$_{MR}$ and SPARE both show a linear decrease in runtime with the increase in core count, however, DCM$_{MR}$ shows better performance as well as scalability than SPARE.

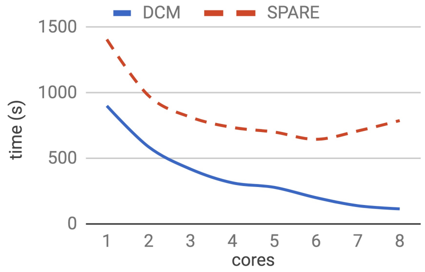

Fig. 9 (A) Execution times for Brinkhoff dataset

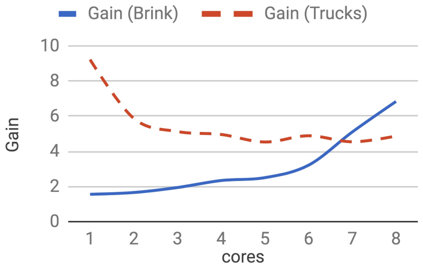

Fig. 10 (A) DCM$_{MR}$ gain over SPARE

Figure 10 shows the performance gain DCM$_{MR}$ shows over the SPARE framework on a logarithmic scale. It can be seen that even though DCM$_{MR}$ performs better than SPARE on the trucks dataset, the performance gain decreases with the increase in the core count. This is the result of the increase in SPARE's performance with the core count. On the Brinkhoff and TDrive datasets, DCM$_{MR}$ shows a linear increase in the performance gain with more cores, achieving higher gains over the TDrive dataset. DCM$_{MR}$ shows gains up to 16 times over SPARE on the TDrive dataset. DCM$_{MR}$ is also able to process large datasets which SPARE fails to do.

9.4 Results: Scale-up Experiments (Hardware Setup B)

This set of experiments was performed to compare the scale-up behaviour of only DCM_{MR} and SPARE. PCCD* was excluded as it being a single threaded algorithm, shows little sensitivity in the scaling experiments. It can be seen in Figures 11 and 13 that both DCM_{MR} and SPARE show linear decrease in runtime with core count, however, DCM_{MR} shows lower execution times and better scale-up performance. On the TDrive dataset (Figure 12), SPARE was unable to finish successfully. Figure 14 shows that DCM_{MR}'s gain over SPARE increases with more cores because of its better scale-up properties. DCM_{MR} shows more than 4 times better performance than the SPARE framework.

To achieve the desired performance of DCM on NUMA architecture, we had to do some fine tuning on the NUMA machine. Following are the recommendations for achieving acceptable scalability and performance out of the DCM algorithm while running on a NUMA machine.

- Make sure the OS has NUMA support enabled.
- Use the JVM with NUMA support.
- Use Linux croups for process isolation.
- Use CPU affinity to prevent the ping-pong effect of moving the process between different CPU.

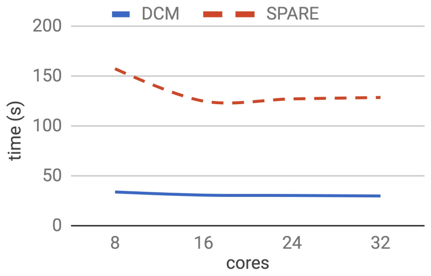

Fig. 11 (B) Execution times for the Trucks dataset

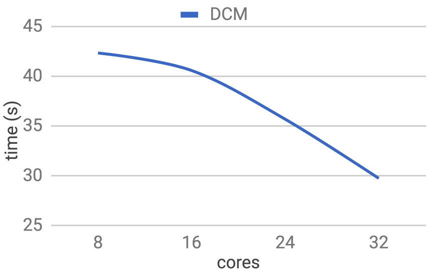

Fig. 12 (B) Execution times for the TDrive dataset. SPARE did not finish in 3 hours

9.5 Results: Scale-out Experiments (Hardware Setup C)

The scale-out experiments were performed using the YARN framework for resource allocation and scheduling. As it can be seen in Figures 15, 16 and 17, DCM_{MR} outperformed SPARE on all datasets and node counts. SPARE was not able to finish on TDrive dataset (Figure. 16). Both SPARE and DCM_{MR} showed linear scalability when run on more nodes. In Figure 18, we can see that DCM_{MR} is up 16 times faster runtime than the SPARE framework.

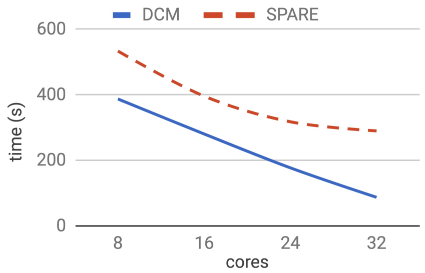

Fig. 13 (B) Execution times for the Brinkhoff dataset

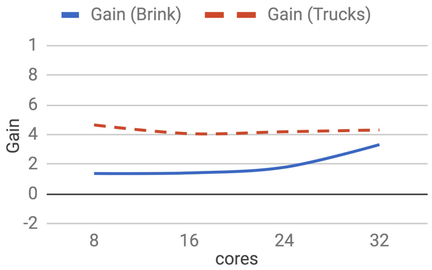

Fig. 14 (B) DCM$_{MR}$ gain over SPARE

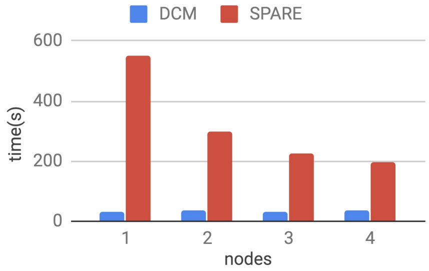

Fig. 15 (C) Execution times for the Trucks dataset

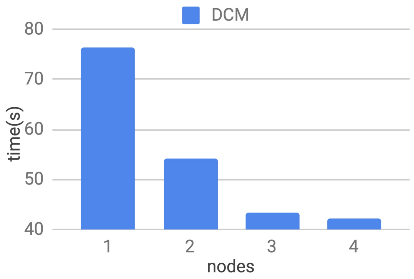

Fig. 16 (C) Execution times for the TDrive dataset (SPARE didn't finish in 3 hours)

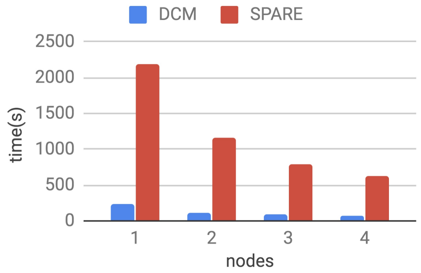

Fig. 17 (C) Execution times for the Brinkhoff dataset

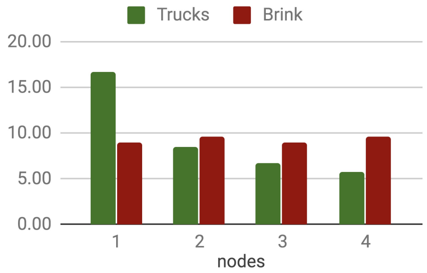

Fig. 18 (C) DCM$_{MR}$ gain over SPARE. SPARE didn't finish on TDrive.

9.6 Results: Experiments DCM$_{MR}$ Analysis

In this set of experiments, we analyze different properties of DCM$_{MR}$'s performance. We also compare it with the baseline sequential algorithm PCCD*. The experiments are performed on hardware setup B to exclude the effects of the network IO and measure DCM's parallelism behaviour.

Data Scalability: Figure 19 and 20 show the performance of DCM$_{MR}$ with respect to the sequential algorithm PCCD* from experiments run on cluster B. The properties of the datasets of the different scales used are shown in

Table 6. In Figure 19, the x-axis denotes the number of cores used for the experiments whereas the y-axis denotes the speedup (speedup percentage with respect to ideal speedup). As it can be seen, DCM$_{MR}$ achieves linear speedup when increasing the number of cores. DCM$_{MR}$ achieves a speedup of up to 13.5 on 16 cores. The performance of DCM$_{MR}$ increases with bigger data sizes. This is because the relative share of Hadoop job setup time in the total execution time becomes less significant for long execution times.

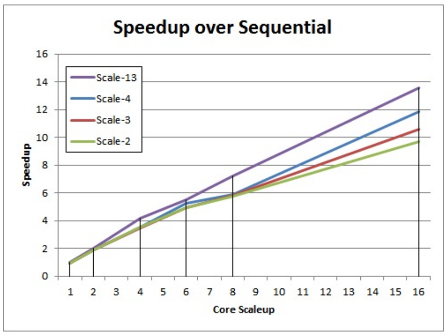

Fig. 19 DCM$_{MR}$ Speedup w.r.t Sequential Algorithm (B)

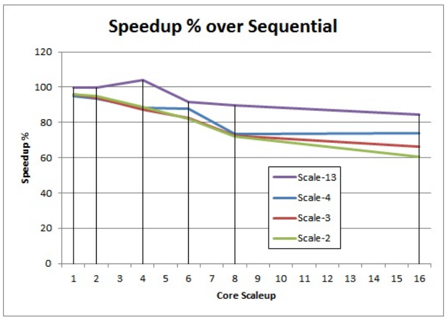

Fig. 20 DCM$_{MR}$ Speedup Percentage w.r.t Theoretical Maximum (B)

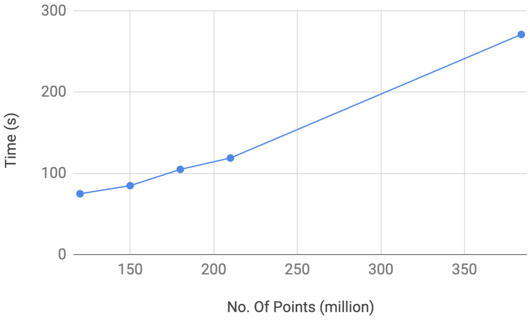

Fig. 21 DCM$_{MR}$ Data Scalability Experiment on Brinkhoff Generator's Datasets

In Figure 20, we can see that the speedup drops to 85 percent of the ideal on 16 cores for scale-13. It is interesting to note that the drop in speedup is smaller for larger data sizes which means that the relative performance of DCM$^*_{MR}$ increases for larger input data sizes. Figure 21 shows the performance of DCM$^*_{MR}$ with the increase in the input data size on setup C. It can be seen that DCM$^*_{MR}$ gives linear scalability with the increase in the data size.

Core-Scalability in NUMA Architecture:

Table 7 Mapping of MapReduce terms to DCM

MapReduce Terms	DCM Terms
MET (Map Execution Time)	DCM$_{part}$ execution time
RET (Reduce Execution Time)	DCM$_{merge}$ execution time
split	partition

Figure 20 shows that the speedup decreases with more cores and drops to approx 83.5 percent on 16 cores. It can be seen that the speedup is almost ideal up to 4 cores after which it starts decreasing. This decrease can be explained with the help of Figure 22. Figure 22 shows that the average Hadoop Mapper Execution Time (MET) increases with more cores. This is caused by the overhead of the NUMA memory controller in coordinating memory access involving multiple cores. The experiment was limited to one machine to isolate the experiments from the network overhead. On the machine, we use one core per NUMA node in an attempt to minimize the effect of resource sharing among the cores in a NUMA node. Note that in each of the experiments, the exact same partitions were processed by the Mappers.

Intermediate False Positive Convoys: Figure 23 shows the percentage of false positive candidate convoys transmitted by DCM$_{part}$ during our first set of experiments. It can be clearly seen that the percentage of false local convoys is around 3 percent which is not high when compared to the total number of global convoys. For scales 1 and 10, there are no false positives. This is because the input data fits in a single HDFS block and are thus not partitioned. Therefore a single DCM$_{part}$ instance runs on the whole input data, producing only closed convoys.

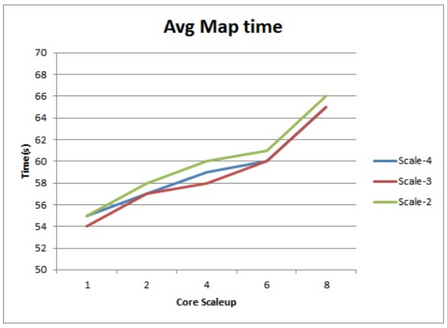

Fig. 22 Effect of increase in the number of cores on avg MET (B)

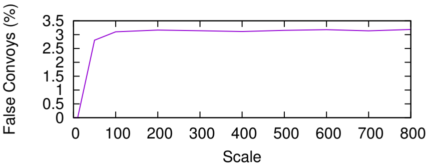

Fig. 23 DCM$_{MR}$ Extra Candidate Convoys

Effect of Partition Size: Table 7 shows the mapping between some terms in the MapReduce domain to our algorithmic terms. We use these terms interchangeably depending on the context.

Figures 24, 25 and 26 show the effect of increasing the number of partitions (splits) of input data or in other words, decreasing the partition sizes, on execution times. Figure 24 shows that the average Hadoop Mapper (DCM$_{part}$) execution time decreases when the partitions become smaller (more number of splits). The decrease is sharp for the number of splits up to 150 and then it becomes gentle. It is obvious to have lower Map execution time for smaller partitions however the effect is almost negligible when we move from 200 splits to 450 splits. To understand that, we need to look into how DCM$^*_{MR}$ treats different partition sizes.

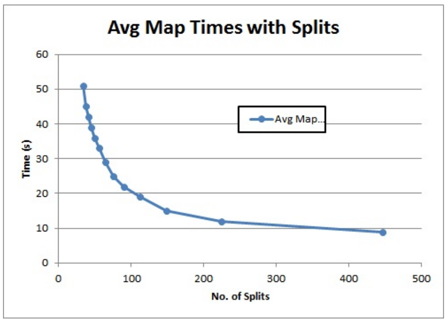

Fig. 24 Effect of Increase in Splits on Average Map Execution Times (B)

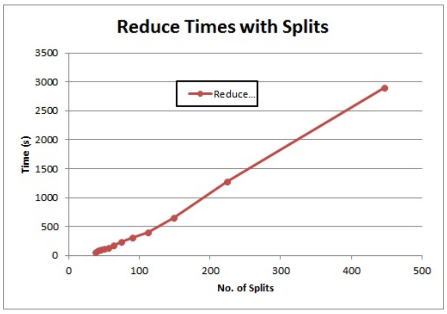

Fig. 25 Effect of Increase in Splits on Reduce Execution Times (B)

Figure 25 shows the effect of increase in the number of splits on Reducer execution time (RET). As it can be seen, RET increases linearly with the increase in the number of splits. The number of intermediate open convoys increases with the increase in the number of time-block owning splits/partitions. This causes the Reducer to process more data. As DCM$^*_{MR}$ runs DCM$_{merge}$ in the Reducer which is the sequential part of the algorithm, the total execution time of the algorithm also increases linearly with RET. This is obvious from the Figure 26.

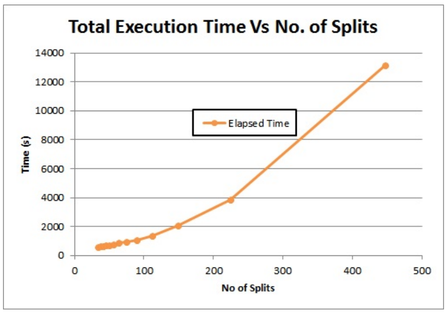

Fig. 26 Effect of No. of Splits on Total Execution Time for Figure 4 (B)

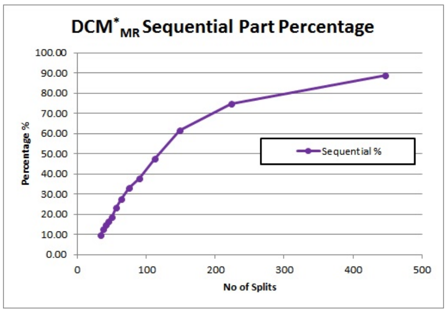

Fig. 27 Effect of No. of Splits on Sequential Part of DCM$^*_{MR}$(B)

Figure 27 shows the proportion of the sequential part of DCM^*_{MR} with the increase in the number of partitions. The percentage of the sequential part on 32 cores grows from over 9% for 32 splits to 89% for 447 splits. The higher the number of splits, more data the Reducer will have to process, thus making the sequential part of the algorithm larger. Hence, ideal number of partitions is approx N^c and idea partition size is $|DB|/N^c$ as shown in section 8.

Job Setup Costs: Figure 19 shows that the speedup is higher for larger data sizes. One reason for higher speed up is that with the increase in data size, the proportion of job setup cost with respect to the total execution time becomes smaller. Job setup cost involves job submission to the Hadoop cluster, starting up the Application Master JVM, resource negotiation with YARN's Resource Manager and starting up of YARN Container JVMs for task execution on each physical machine etc. Job setup cost is almost constant for each job running on the same number of cores and machines.

Job Running Costs: In addition to job set up costs, YARN incurs some costs for running the job as well. When the job is setup, the Node Manager executes the Mapper tasks in the YARN containers alloted for the job. When a Mapper finishes and a new Mapper needs to be executed, the same container cannot be used. The existing container is shut down and a new container JVM is launched. This process is repeated for each partition. Earlier versions of Hadoop, i.e., version 1.* supported JVM reuse which helped in reducing the running costs of a job. This feature is unfortunately not available in Hadoop version 2.* using YARN. Therefore, to reduce the job running costs, the number of partitions should be equal to the number of available cores. This way, there will be one Mapper per core which will be executed by one YARN container saving multiple JVMs starting costs.

MET Distribution: Figure 28 shows the spread and groupings of METs for different number of cores when Hadoop's default partition/split size is used. We can see that not only the METs are widely spread over time but they are also not part of a single cluster. These results are also true for custom split sizes and depend on the distribution of time-blocks within partitions and the distribution of points between time-blocks. We can clearly see an unbalanced cost distribution between Mappers. Even though the DCM algorithm has nice parallelization properties, skewness in the data or bad partitioning can badly affect the performance of the algorithm. We also see a comparatively very small MET which is an outlier and represents the processing time of the last partition (HDFS block) which is smaller than the rest of the partitions (HDFS blocks).

Figure 29 shows the histogram of METs which somewhat resemble with a normal distribution. The mean and standard deviation of the distributions increases with the increase in the number of cores. This increase is a strong evidence of decrease in the performance of NUMA nodes because of memory sharing/context switching and inter node memory IO bottlenecks when more cores are used.

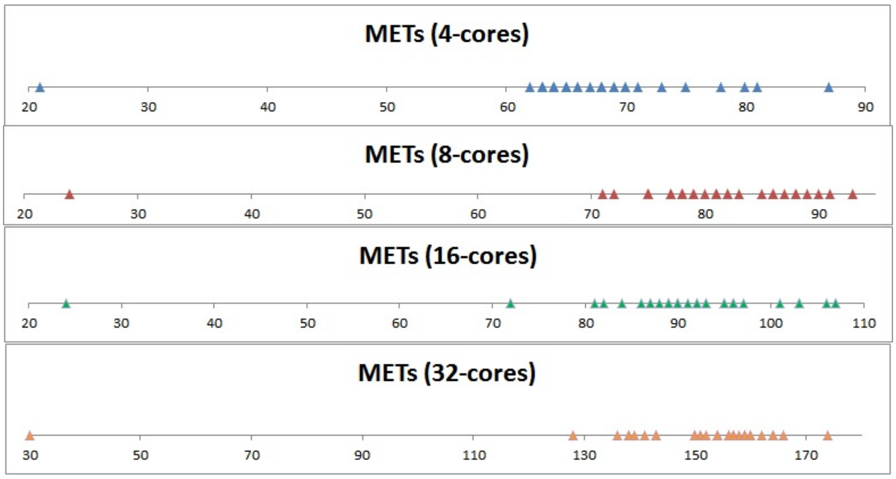

Fig. 28 Map Execution Times Clusters (B)

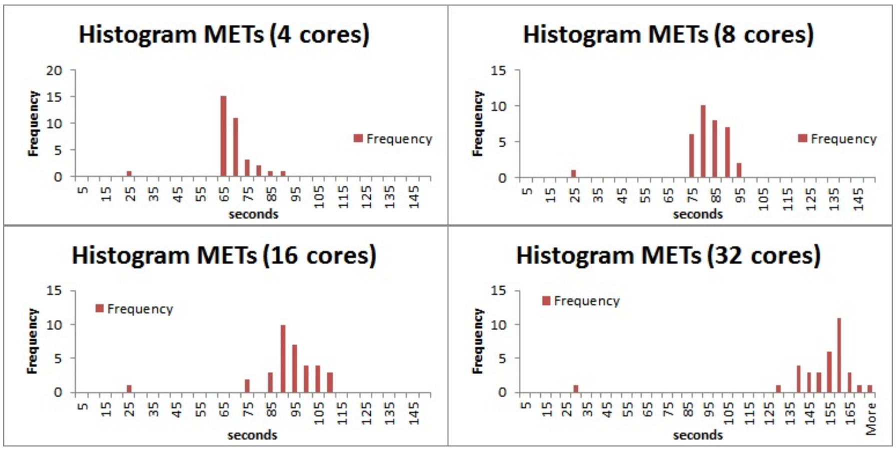

Fig. 29 Histogram for Map Execution Times (B)

Factors Affecting Speedup: The execution time of DCM$_{MR}$ and DCM$^*_{MR}$ running on a 4 nodes cluster is 3.6 times lower than its execution time on a single node. There are certain factors which prevent DCM$_{MR}$ from giving N times performance increase with N nodes which we analyze in the following.

Partial Parallelization: DCM$_{MR}$ parallelizes density based clustering and a major part of convoy mining but the final merge is performed on a single node achieving parallelization of around 90% , i.e., around 90% of the time is taken by Map phase which executes in parallel.

Partial Cluster Utilization: When the number of partitions p is not a multiple of the number of nodes N, the cluster is underutilized, e.g., for $p = 25$ and $N = 24$, processing the first 24 partitions will fully utilize the cluster after which the last partition will be processed with a utilization factor of 1/24 thus reducing the overall speedup.

Network Costs: DCM$_{MR}$ needs to collect the local results of DCM$_{part}$ from cluster nodes over the network, the cost of which is significantly higher than the cost of memory IO utilized by PCCD*.

Disk IO: Unlike PCCD*'s in-memory computations, the Hadoop writes the output of each Map and Reduce phase to disk. Disk IO is many times more expensive than memory IO and affects the performance of DCM$_{MR}$. Disk IO costs include the costs of disk seeks, scans, serialization and deserialization operations.

Sorting Costs: The output of the Mappers is partitioned and sorted in the shuffle phase. DCM$_{merge}$ needs the input local convoys to be sorted with respect to the merge time of convoys. Shuffle in Hadoop is a costly process with cost complexity of $\mathcal{O}(nlog(n))$.

10 Conclusion and Future Work

Convoy pattern mining is computationally expensive and existing algorithms do not scale up to the huge amounts of movement data. In this paper we propose a generic, framework independent, and highly scalable distributed convoy pattern mining algorithm called DCM (Distributed Convoy Mining) which outperforms existing algorithms by a high margin. We choose temporal partitioning over spatial partitioning because of its merits for convoy pattern mining. We performed detailed theoretical analysis of the DCM algorithm and various factors affecting it. We modeled the effect of number of partitions/partition size and showed that a bad choice of the number of partitions can result in low cluster utilization and severely harm the performance of the parallel part of the algorithm thus affecting the overall run-time.

We implement our generic algorithm in the Hadoop MapReduce framework and using the KD-Tree index for speeding up the clustering phase. We test our algorithm extensively on the real-world and synthetic datasets with sizes varying from a few million records up to 384 million records and number of objects varying from a couple of hundreds to 6.5 million. The results show that DCM$_{MR}$ can process large datasets which cannot be processed by SPARE which is the state of the art co-movement pattern mining framework. For other datasets DCM$_{MR}$ outperforms SPARE and shows gains of up to 16 times. We validate the high performance and scalability of our algorithm w.r.t both the cluster and dataset sizes. We analyze the performance of DCM on NUMA architecture machines and show that the performance of individual cores decrease with an increase in NUMA network utilization, thus, decreasing the scalability of DCM. We show the effect of the partition size on the sequential part as well as the total execution time of the algorithm.

In the future, we would like to design a convoy mining algorithm that works with spatial as well as a hybrid spatio-temporal partitioning strategy. For the high density datasets in which billions of points correspond to a single timestamp, spatial partitioning can be of help. We would also like to test the performance of DCM on other frameworks, e.g., Apache Flink and Apache Spark.

References

1. Aung, H.H., Tan, K.L.: Discovery of evolving convoys. In: International Conference on Scientific and Statistical Database Management, pp. 196–213. Springer (2010)
2. Brinkhoff, T.: Generating network-based moving objects. In: Scientific and Statistical Database Management, 2000. Proceedings. 12th International Conference on, pp. 253–255. IEEE (2000)
3. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
4. Chen, T.S., Chang, C.Y.: Skewed data partition and alignment techniques for compiling programs on distributed memory multicomputers. The Journal of Supercomputing **21**(2), 191–211 (2002)
5. Dai, B.R., Lin, I., et al.: Efficient map/reduce-based dbscan algorithm with optimized data partition. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pp. 59–66. IEEE (2012)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
7. Douglas, D.H., Peucker, T.K.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: The International Journal for Geographic Information and Geovisualization **10**(2), 112–122 (1973)
8. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Kdd, vol. 96, pp. 226–231 (1996)
9. Fan, Q., Zhang, D., Wu, H., Tan, K.L.: A general and parallel platform for mining co-movement patterns over large-scale trajectories. Proceedings of the VLDB Endowment **10**(4), 313–324 (2016)
10. Gudmundsson, J., van Kreveld, M.: Computing longest duration flocks in trajectory data. In: Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems, pp. 35–42. ACM (2006)
11. He, Y., Tan, H., Luo, W., Feng, S., Fan, J.: Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. Frontiers of Computer Science **8**(1), 83–99 (2014)
12. Hua, K.A., Lee, C.: Handling data skew in multiprocessor database computers using partition tuning. In: VLDB, pp. 525–535. Citeseer (1991)
13. Jeung, H., Shen, H.T., Zhou, X.: Convoy queries in spatio-temporal databases. In: 2008 IEEE 24th International Conference on Data Engineering, pp. 1457–1459. IEEE (2008)
14. Jeung, H., Yiu, M.L., Zhou, X., Jensen, C.S., Shen, H.T.: Discovery of convoys in trajectory databases. Proceedings of the VLDB Endowment **1**(1), 1068–1080 (2008)
15. Kalnis, P., Mamoulis, N., Bakiras, S.: On discovering moving clusters in spatio-temporal data. In: International Symposium on Spatial and Temporal Databases, pp. 364–381. Springer (2005)
16. Kwon, Y., Ren, K., Balazinska, M., Howe, B., Rolia, J.: Managing skew in hadoop. IEEE Data Eng. Bull. **36**(1), 24–33 (2013)
17. Lacerda, T., Fernandes, S.: Scalable real-time flock detection. In: Global Communications Conference (GLOBECOM), 2016 IEEE, pp. 1–7. IEEE (2016)

18. Naserian, E., Wang, X., Xu, X., Dong, Y.: Discovery of loose travelling companion patterns from human trajectories. In: High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on, pp. 1238–1245. IEEE (2016)
19. Orakzai, F., Calders, T., Pedersen, T.B.: Distributed convoy pattern mining. 17th IEEE International Conference on Mobile Data Management. (2016)
20. Orakzai, F., Devogele, T., Calders, T.: Towards distributed convoy pattern mining. In: Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '15, pp. 50:1–50:4. ACM, New York, NY, USA (2015). DOI 10.1145/2820783.2820840. URL http://doi.acm.org/10.1145/2820783.2820840
21. Patwary, M.M.A., Palsetia, D., Agrawal, A., Liao, W.k., Manne, F., Choudhary, A.: A new scalable parallel dbscan algorithm using the disjoint-set data structure. In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, pp. 1–11. IEEE (2012)
22. Tang, L.A., Zheng, Y., Yuan, J., Han, J., Leung, A., Hung, C.C., Peng, W.C.: On discovery of traveling companions from streaming trajectories. In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on, pp. 186–197. IEEE (2012)
23. Vieira, M.R., Bakalov, P., Tsotras, V.J.: On-line discovery of flock patterns in spatio-temporal data. In: Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems, pp. 286–295. ACM (2009)
24. Wang, D., Joshi, G., Wornell, G.: Efficient task replication for fast response times in parallel computation. In: ACM SIGMETRICS Performance Evaluation Review, vol. 42, pp. 599–600. ACM (2014)
25. Yoon, H., Shahabi, C.: Accurate discovery of valid convoys from moving object trajectories. In: ICDM workshops, pp. 636–643 (2009)
26. Yuan, J., Zheng, Y., Xie, X., Sun, G.: Driving with knowledge from the physical world. In: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 316–324. ACM (2011)
27. Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., Huang, Y.: T-drive: driving directions based on taxi trajectories. In: Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems, pp. 99–108. ACM (2010)
28. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 2–2. USENIX Association (2012)

Appendices

A Scalability on the NUMA Architecture

NUMA (Non Uniform Memory Access) systems are low-cost multi-processor platforms that support large numbers of processors on a single board. Faster CPUS are generally constrained by the memory bandwidth under memory-intensive workload. Symmetric multiprocessing (SMP) systems use a shared bus to connect processors, thus, many processors have to compete for memory bandwidth. The NUMA architecture solves this problem by connecting several low-end processor nodes each having its own cache and memory, using a high-speed connection. Each node has a memory controller which allows it to use memory on all other nodes in addition to its own memory, thus abstracting the memory as a single image. When a processor requests data from a memory location that does not exist in its local memory, the data is transfered over the NUMA connection, which is slower than the connection between the processor and its local memory. Thus, memory access time is not uniform and varies depending upon if the access is local or remote.

In NUMA systems, cache coherence problem occurs when two or more processors access the same shared data. If one processor modifies its copy of the data, the copies of this data in the cache of other processors will become stale. ccNUMA (cache coherent NUMA) machines ensure that a processor accessing a memory location receives the most up-to-date version of the data. Cache coherence can be ensured either in software or hardware, however software approaches tend to be slower than the hardware ones.

We analysed the performance of DCM$_{MR}$ on AMD Opteron based NUMA machines. Figure 30 shows the architecture of the AMD Opteron 6300 series processors. The processor has two NUMA nodes connected with a HyperTransport bus. Each node has 8 cores. The cores are arranged in 4 pairs such that each pair shares a Floating Point Unit (FPU) and an L2 cache of 2MB. The pairs are connected to each other by the Crossbar Switch which connects to the HT bus through an HT interface. Each node has its own memory controller with 2 channels. Each channel supports memory up to 32 GB.

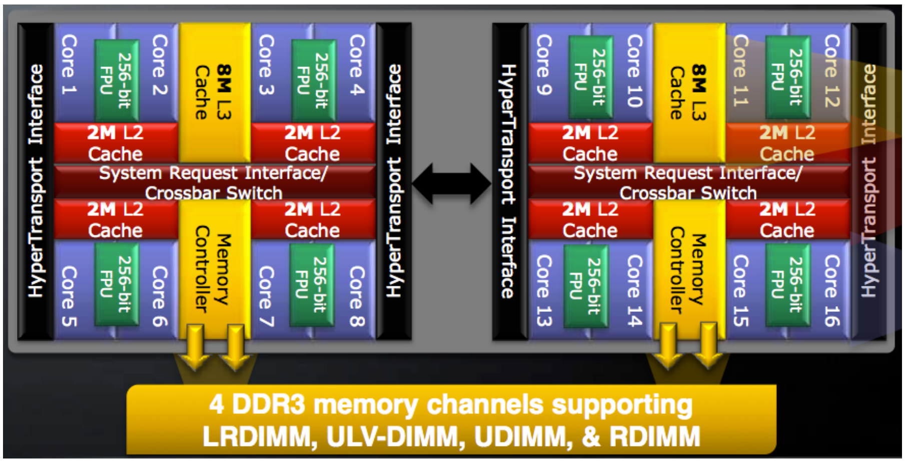

Fig. 30 AMD Opteron 6300 series Processor Architecture[5]

Figure 31 shows the architecture of the AMD Opteron 6300 series quad-processor cc-NUMA system which we used for one set of our experiments. The system consists of 4 AMD Opteron 6376 processors (Figure 30) interconnected through HT buses. The system has 512 GB of memory (128 GB per processor, 64 GB per NUMA node). If a processor core is the first one to request a memory page, it is mapped to the memory of the node to which the core belongs (first touch policy). A NUMA aware OS tries to keep the threads running always in the same core pair because they share the same L2 cache. Moving a thread to another core-pair will cause a performance degradation because of cache invalidation. A thread gets further performance hit if it is moved to another node because it needs to get data from a remote node's memory. Therefore using multiple cores for running a process using context switching although increases the performance but the increase might not be linear depending upon the location of the core the process is moved to.

If an algorithm accesses all of its data from the memory, ccNUMA increases memory bandwidth at a ratio effectively the same as the number of NUMA nodes. In our case, it is expected to have 8 times the memory bandwidth of an SMP machine but it does not necessarily mean that the performance of an algorithm will scale linearly with increase in the number of cores because of the performance bottlenecks explained above. The following steps are required from a NUMA-aware OS for optimal NUMA performance:

[5] http://beagle.ci.uchicago.edu/technical-specification

- Processes should be scheduled on cores as close as possible to the memory that contains its data.
- OS should maintain a queue per node
- Memory allocation for a process should be in the memory of a single node
- All child processes should be scheduled on the same node during the lifetime of the parent process

The two most common policies supported by the Linux kernel are `NODE LOCAL` and `INTERLEAVE` [6,7]. In `NODE LOCAL` mode, an allocation occurs from the memory node local to where the code is currently executing where as in the `INTERLEAVE` mode, allocation occurs round-robin. The `INTERLEAVE` policy is used to distribute memory accesses for data structures that may be accessed from multiple processors in the system in order to have an even load on the interconnect and the memory of each node.

The memory management policies of the OS work best for the general cases and not for a specific application with a different memory access behaviour. When the memory load of a NUMA system increases, its memory management overhead increases, thus resulting in the overall degraded performance. Therefore the best approach is to have an application do the management itself. Hadoop runs in Java Virtual Machines (JVMs) which come with support for NUMA but Hadoop itself is not NUMA aware. Thus, an algorithm running on Hadoop on a NUMA system shows lower scalability in terms of number of cores when compared to its execution on a cluster of SMP machines with the same number of cores.

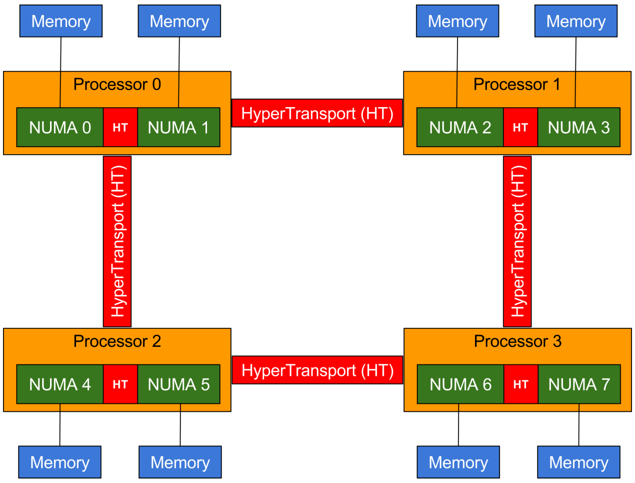

Fig. 31 AMD Opteron 6300 series Multi-node Architecture

[6] http://queue.acm.org/detail.cfm?id=2513149

[7] https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt