Parallel All Pairs Similarity Search

Amit Awekar, and Nagiza F. Samatova[1]
acawekar@ncsu.edu, samatovan@ornl.gov
North Carolina State University, Raleigh, NC
Oak Ridge National Laboratory, Oak Ridge, TN
[1]Corresponding Author

***Abstract*— This paper presents the first scalable parallel solution for the All Pairs Similarity Search ($APSS$) problem, which involves finding all pairs of data records that have a similarity score above the specified threshold. With exponentially growing datasets and modern multi-processor/multi-core system architectures, serial nature of all existing $APSS$ solutions is the major rate limiting factor for applicability of $APSS$ to large-scale real-world problems and calls for parallelization. Our proposed $index\ sharing$ technique divides the $APSS$ computation into independent searches over the central inverted index shared across all processors as a read-only data structure and achieves linear speed-up over the fastest serial $APSS$ algorithm in shared memory environment. We demonstrate effectiveness of our solution over four large-scale real world million record datasets.**

I. Introduction

All Pairs Similarity Search *(APSS)* is the problem of finding all pairs of data records having similirty score above the specified threhsold. Similarity between two records is defined via well known similarity measures, such as the cosine similarity or the Tanimoto coefficient. Many Business and scientic applications like search engines, and systems biology frequently solve the $APSS$ problem over high diemnsional datasets having several millions or billions of records.

The nature of the existing $APSS$ solutions is compute- as well as data-intensive. Given a dataset with n data records in a d dimensional space, where $n << d$, existing $APSS$ algorithms compute $O(n^2)$ similarity scores, while searching through $O(n*d)$ size inverted index that maps each dimension to a list of data records having a non-zero projection along that dimension.

Existing solutions for $APSS$ are all limited to serial algorithms [1], [2], [3], [4], [5], [6]. The compute- and data-intensive nature of $APSS$ is a rate limiting factor for the $APSS$ applicability to large-scale real-world problems and calls for alternative approaches. Processor clock rates are not expected to increase dramatically in the near future [7]. With the emergence of shared memory multi-processor, multi-core architectures, parallel algorithms that take advantage of such emerging architectures are a promising strategy. Throughout this paper, we will use the term *processor* to refer to a single processor or a processing core within a multi-core processor, unless stated otherwise. Inspired by the success of parallel computing in dealing with large-scale problems [8], [9], [10], we explore parallelization to further speed-up $APSS$ computation.

Parallel algorithms for $APSS$ should enable processing of large datasets in a reasonable amount of time. Web-based applications like search engines, online social networks, and digital libraries are increasingly dealing with more massive datasets [11], [12]. Without scalable parallel $APSS$ algorithms, it will likely not be practical to run $APSS$ over some of these applications' datasets, which are growing at an exponential rate [13], [14].

A scalable, parallel solution for $APSS$ will effectively help design scalable, parallel solutions for important data mining tasks like clustering and collaborative filtering that use $APSS$ as their underlying operator. Middlewares like pR [15] can use parallel solution for $APSS$ to speed-up statistical analysis algorithms.

The compute- and data-intensive nature of the $APSS$ problem poses the following technical challenges for its parallel solution:

- The inverted index is shared across all processors and updated incrementally.
- The huge size of the dataset and of the inverted index makes data transfers between processors prohibitively expensive.

Our proposed *index sharing* technique addresses these challenges by parallelizing the $APSS$ computation into independent searches over the central inverted index, which is shared as a read-only data structure across all processors. Index sharing builds the whole central inverted index before starting the search for *matching pairs*, unlike existing $APSS$ algorithms that incrementally build the inverted index while searching for *matching pairs*. Each processor keeps and updates its own copy of index metadata of reasonably small size, resulting in slightly larger memory footprint. A subset of data records is assigned to each processor to find the corresponding matching pairs. We explore various static and dynamic strategies for distributing data records among processors. We empirically evaluate the performance of the proposed index sharing technique using four real-world million record datasets described in Section VI.

To the best of our knowledge, this is the first work that explores parallelization for $APSS$. We propose the following contributions:

- We develope the index sharing technique to parallelize the APT algorithm [6], which is the fastest serial $APSS$ algorithm (Please, refer to Figures 1 and 2).

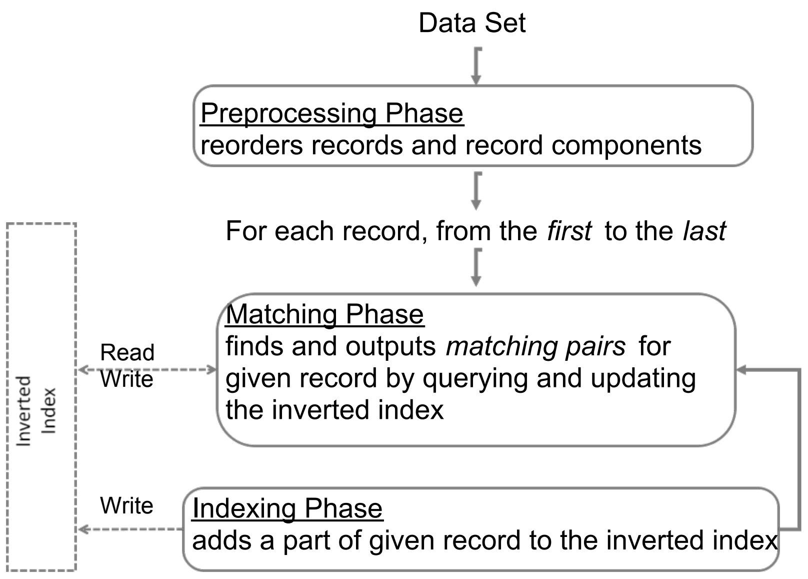

Data Set
Preprocessing Phase
reorders records and record components
For each record, from the first to the last
Matching Phase
finds and outputs matching pairs for given record by querying and updating the inverted index
Read
Write
Inverted Index
Write
Indexing Phase
adds a part of given record to the inverted index

Fig. 1: Unifying Framework for Recent Exact $APSS$ Algorithms

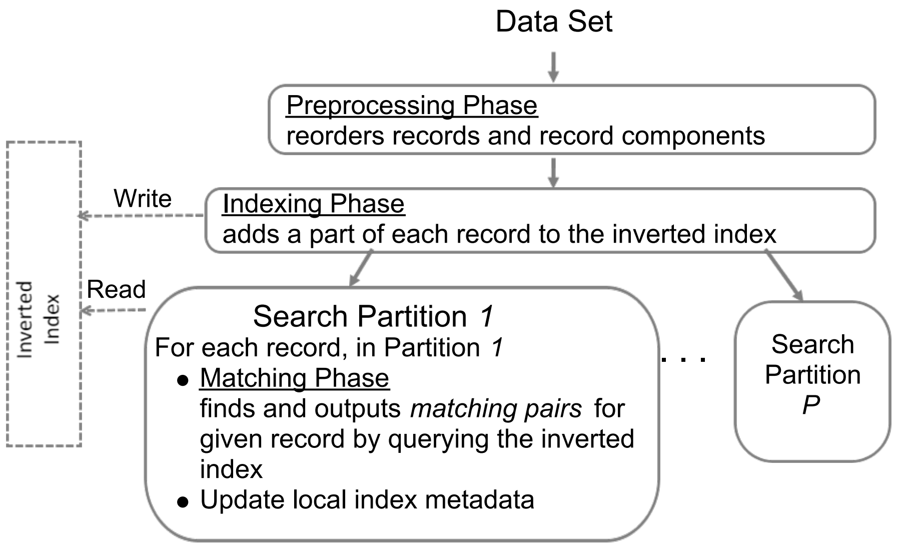

Data Set
Preprocessing Phase
reorders records and record components
Write
Indexing Phase
adds a part of each record to the inverted index
Inverted Index
Read
Search Partition 1
For each record, in Partition 1
• Matching Phase
finds and outputs matching pairs for given record by querying the inverted index
• Update local index metadata
. . .
Search Partition P

Fig. 2: Proposed Parallel $APSS$ Solution

- Our index sharing based parallel $APSS$ algorithm achieves linear speed-up over the APT algorithm in a shared memory environment.
- We provide a scalable solution to perform $APSS$ over large datasets in a reasonable time.

II. Definitions and Notations

In this section, we define the $APSS$ problem and other important terms referred throughout the paper.

Definition 1 (*All Pairs Similarity Search*): The all pairs similarity search ($APSS$) problem is to find all pairs (x, y) and their exact value of similarity $sim(x, y)$ such that $x, y \in V$ and $sim(x, y) \geq t$, where

- V is a set of n real valued, non-negative, sparse vectors over a finite set of dimensions D; $|D| = d$;
- $sim(x, y) : V \times V \rightarrow [0, 1]$ is a symmetric similarity function; and
- t, $t \in [0, 1]$, is the similarity threshold.

Definition 2 (*Inverted Index*): The inverted index maps each dimension to a list of vectors with non-zero projection along that dimension. A set of all d lists $I = \{I_1, I_2, \ldots, I_d\}$, i.e., one for each dimension, represents the inverted index for V. Each entry in the list has a pair of values (x, w) such that if $(x, w) \in I_k$, then $x[k] = w$. The inverse of this statement is not necessarily true, because some algorithms index only a part of each vector.

Definition 3 (*Candidate Vector* and *Candidate Pair*): Given a vector $x \in V$, any vector y in the inverted index is a candidate vector for x, if $\exists\ j$ such that $x[j] > 0$ and $(y, y[j]) \in I_j$. The corresponding pair (x, y) is a candidate pair.

Definition 4 (*Matching Vector* and *Matching Pair*): Given a vector $x \in V$ and the similarity threshold t, a candidate vector $y \in V$ is a matching vector for x, if $sim(x, y) \geq t$. We say that y matches with x, and vice versa. The corresponding pair (x, y) is a matching pair.

During subsequent discussions we assume that all vectors are of unit length ($||x|| = ||y|| = 1$), and the similarity function is the cosine similarity. In this case, the cosine similarity equals the dot product, namely:

$$sim(x, y) = cos(x, y) = dot(x, y).$$

Our algorithm can be extended to the Tanimoto coefficient and other similarity measures using simple conversions derived by Bayardo *et al.* [3].

III. Previous Work

All existing algorithms for $APSS$ are serial in nature. Previous work on $APSS$ can be divided into two main categories: heuristic and exact.

Main techniques employed by heuristic algorithms are hashing, shingling, and dimensionality reduction. Charikar [16] defines a hashing scheme as a distribution on a family of hash functions operating on a collection of vectors. For any two vectors, the probability that their hash values will be equal is proportional to their similarity. Fagin *et al.* [17] combined similarity scores from various voters, where each voter computes similarity using the projection of each vector on a random line. Broder *et al.* [18] use shingles and discard the most frequent features.

Recent inverted index based exact algorithms for $APSS$ have outperformed the heuristic algorithms. These exact algorithms share a common three-phase framework of:

- data preprocessing: sorting data records and computing summary statistics;
- pairs matching: computing similarity between selective record pairs; and
- record indexing: adding a part of data record to an indexing data structure.

The preprocessing phase reorders data records and record components using various attributes such as: the number of components in a data record or the maximum component value within a data record. Then, the matching phase identifies, for a given record, corresponding pairs with the similarity above the specified threshold by querying the inverted index. The matching phase also removes redundant entries from the inverted index. The indexing phase then adds a part of the given data record to the inverted index. The matching phase dominates the computing time of $APSS$, and the time spent during preprocessing and indexing is negligible.

The APT algorithm [6] is also based on the common framework described above and is the fastest serial algorithm for $APSS$. Please, refer to Figure 1 for an overview of the APT algorithm.

IV. Index Sharing

The index sharing technique is based on parallelizing the matching phase of $APSS$ into independent searches over *the central inverted index which is shared across all processors* as a read-only data structure. In contrast, *the data records are partitioned among processors* to perform the matching phase. We explore both static and dynamic partitioning strategies. Each processor performs the matching phase independently of other processors using the central inverted index. While finding the matching pairs for a given record using the central inverted index, the procedure used by the index sharing technique is the same as the procedure used by the APT algorithm. Please, refer to Figure 3 for an overview of the index sharing technique.

Read and write access to the inverted index during the matching phase is the major bottleneck in parallelizing the APT algorithm (please, refer to Figure 1). This bottleneck arises because of the following two reasons:

1) APT algorithm adds a given data record to the inverted index only after performing the matching phase for that record. Thus, new entries are added to the inverted index during the matching phase.
2) The matching phase in the APT algorithm updates the inverted index during each search by discarding redundant entries from the inverted index.

If the matching phase in a parallel $APSS$ algorithm requires read and write access to the central inverted index, then each processor will require exclusive access to the inverted index, resulting in *synchronization overheads*. Index sharing overcomes this bottleneck by building the inverted index before starting the matching phase and by replicating the reasonably small size index metadata across all processors.

To perform the matching phase for a given data record, all the data records with the ids prior to the given record's id must be present in the inverted index. Hence, the APT algorithm cannot be adopted directly to perform search for multiple data records simultaneously. Index sharing overcomes this limitation by building the whole inverted index before starting the matching phase (please refer to Figure 2).

A. Index Metadata Replication

Index metadata is the set of start offset values maintained, one for each list of data record ids in the inverted index, indicating the front of the list. Index sharing replicates the index metadata across all processors to eliminate the need

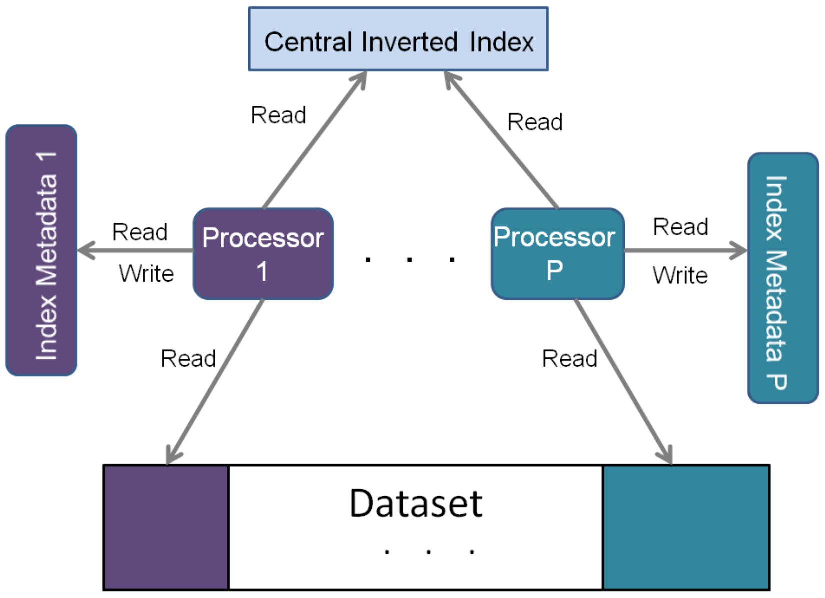


Fig. 3: Overview of the Index Sharing Technique

for any synchronization among processors while performing the matching phase. For a dataset with n data records in a d dimensional space, the inverted index contains $O(n * d)$ entries, while the size of the metadata is only $O(d)$. Compared to the size of the inverted index, the size of the metadata is reasonably small and grows linearly with the number of dimensions.

The matching phase of the APT algorithm requires write access to the inverted index to discard redundant entries from the inverted index. The preprocessing phase in the APT algorithm sorts data records in the decreasing order of the maximum value of any component within the record. Using this sort order, the APT algorithm derives a lower bound on the size of data records in the inverted index to match with any of the remaining data records. While performing the matching phase for a given record, the entries that correspond to data records not satisfying the lower bound on their size are discarded from the inverted index.

For time efficiency purposes, the APT algorithm does not actually remove the redundant entries from the inverted index, but only ignores them using the index metadata. The APT algorithm uses arrays for representing lists in the inverted index. Deleting an element from the beginning of a list will have linear time overhead. Instead of actually deleting such entries, the algorithm simply ignores these entries by removing them from the front of the list. The start offset corresponding to an inverted list array is incrementally advanced as entries are removed from the front.

Index sharing replicates the index metadata across all processors to eliminate the need for synchronization between processors while performing the matching phase. Each processor updates its local index metadata after performing the matching phase for every data record assigned to it.

V. Load Balancing Strategies

The goal of the index sharing technique is to divide the computation workload of the matching phase roughly equally across all processors. We consider two static partitioning strategies (Block and Round-Robin) and a dynamic load balancing strategy.

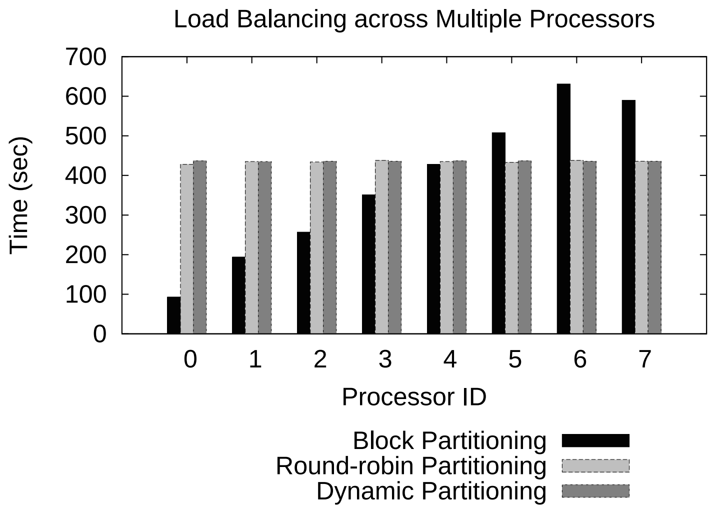
A. *Block Partitioning*

The block load balancing strategy assigns a contiguous block of data records to each processor. The time required to perform the matching phase for a given data record increases as $APSS$ proceeds from the beginning of the dataset to the end. This variation arises because the preprocessing phase puts short data records, i.e. records with fewer number of non-zero components at the beginning of the dataset. Compared to short data records, longer data records require more time to perform the matching phase because they generate comparatively more candidate pairs for evaluation.

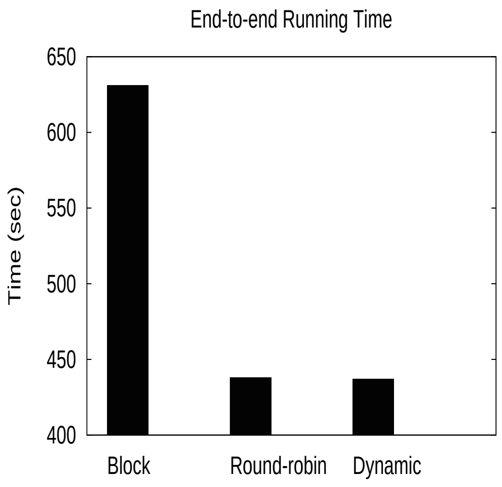
Due to the variation in the time required for the matching phase of data records, assigning equal number of contiguous data records to each processor will likely create severe work imbalance among the processors. The block load balancing strategy tries to compensate this imbalance by assigning an equal number of components to each processor. If short data records are assigned to a processor, then that processor will have more number of data records assigned than a processor with longer data records assigned.

B. *Round-Robin Partitioning*

The block load balancing strategy assigns all short data records to initial processors and all longer data records to later processors, resulting in a severe imbalance in the distribution of the computation workload of the matching phase across



(a) Different Processors' Execution Times



(b) End-to-end Running Time

Fig. 4: Comparison of Various Load Balancing Strategies

various processors. Please, refer to Figure 4 for an example of this imbalance. This example was generated while running an index sharing based parallel $APSS$ algorithm using block load balancing strategy for the Orkut dataset.

Work imbalance induced by the block load balancing strategy can be reduced by assigning data records to each processor in a Round-Robin fashion. If there are P processors, then any consecutive P data records in the dataset are assigned to a different processor by the Round-Robin load balancing strategy. Please, refer to Figure 4 for an example of the performance improvement achieved in evenly distributing the computation workload of the matching phase by the Round-Robin strategy over the block strategy.

C. Dynamic Partitioning

Dynamic partitioning strategy aims at maximizing the processor utilization efficiency by dynamically assigning a small batch of data records to a processor as soon as the corresponding processor finishes the previous batch. As a result, all processors are expected to finish their computation almost the same time.

Contrary to the general experience in parallel computing, dynamic load balancing strategy performs only marginally better than the Round-Robin load balancing strategy which is a static strategy. The specific sort order of data records is the reason for the exceptionally good performance of Round-Robin strategy.

VI. Index Sharing Performance Evaluation

We performed experiments on four real-world million record datasets: Medline, Flickr, LiveJournal, and Orkut. Medline dataset comes from the scientific literature collaboration information in Medline indexed papers, while the rest come from popular online social networks: Flickr, LiveJournal, and Orkut. These datasets represent a variety of large-scale web-based applications like digital libraries and online social networks that we are primarily interested in. More detailed description these datasets is available in [19].

The distribution of the vector sizes in these datasets is the power law distribution [20], [6], [3]. These datasets are high dimensional and sparse (please, refer to Table I). The ratio of the average vector size to the total number of dimensions is less than 10^{-4}. All these characteristics are common across datasets generated and used by many large-scale web based applications [4], [3]. Therefore, we expect our index sharing technique to be relevant to other similar datasets as well.

We performed experiments for both the cosine similarity and the Tanimoto coefficient measures. Results for both similarity measures are quite similar. We present results only for cosine similarity for the sake of brevity. The results presented here are an aggregate of experiments preformed by varying the similarity threshold value from 1.0 to 0.5 in decrements of 0.1. The time spent for preprocessing and indexing is negligible as compared to the time spent for the matching phase. In all our experiments, we consider the time required only for the matching phase.

All of our implementations are for shared memory environment and coded in C++. We implement parallelization using the POSIX Pthreads library [21]. We used the standard template library for most of the data structures. We used the dense hash map class from Google^{TM} for the hash-based partial score accumulation [22]. The code was compiled using the GNU gcc 4.2.4 compiler with $-O3$ option for optimization. The experiments were performed on multi-processor as well as multi-core shared memory computers, each with eight processing elements. The code, the datasets, and additional experimental results are available for download on the Web [23].

TABLE I: Datasets Used

Dataset	Number of Records	Total Non-zero Components	Average Size
Medline	1565145	18722422	11.96
Flickr	1441433	22613976	15.68
LiveJournal	4598703	77402652	16.83
Orkut	2997376	223534153	74.57

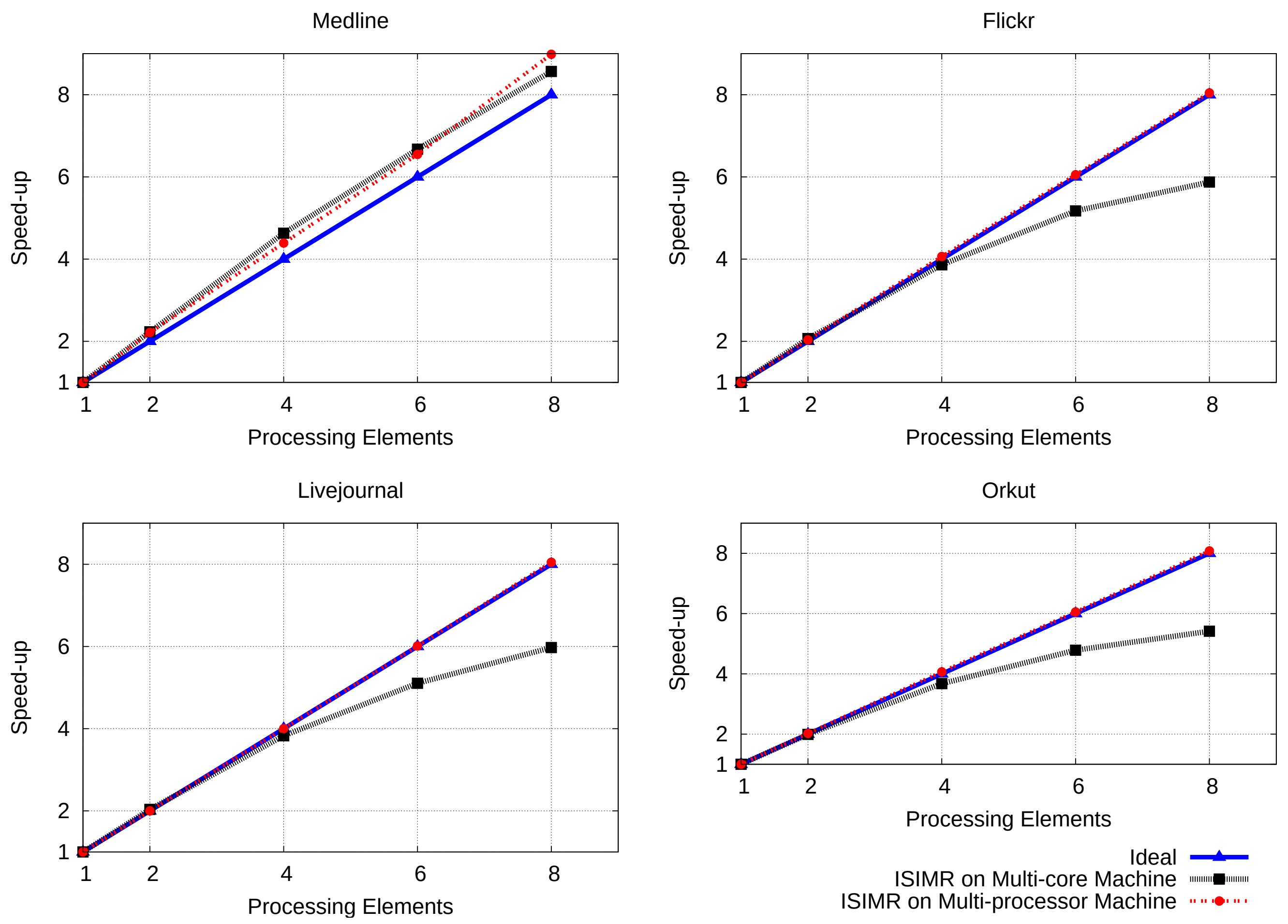


Fig. 5: Speed-up Over APT Algorithm vs. Number of Processing Elements for Index Sharing Technique

As described in Section I, motivation for parallelizing $APSS$ is to create a solution that scales with the number of processing elements as well as with the size of datasets. Therefore, we evaluate the performance of the index sharing technique based on scalability with respect to the number of processing elements and to the number of data records. The index sharing technique achieves ideal performance for both metrics.

For multi-processor environment, index sharing achieves ideal strong scaling behavior, i.e. linear speed-up over the APT algorithm (please, refer to Figures 5. The performance of the index sharing technique degrades in the multi-core environment due to *cache thrashing* and *memory bandwidth* limitation. In our experiments, the size of the datasets and of the inverted index range from few hundred megabytes to multiple gigabytes. $APSS$ algorithms access the inverted index and the dataset randomly, resulting in cache thrashing. Multi-core environment has multiple processing cores, but they still share the bus connection to the shared memory. When more cores start competing for memory access, the index sharing technique performance degrades. Thrashing effect is more visible for larger datasets like Orkut, while linear speed-up is maintained for smaller datasets, like Medline.

Scalability of index sharing with respect to variations in dataset sizes is plotted in Figure 6. The performance of the index sharing technique remains consistent. This result suggests that the index sharing technique will likely scale well with other large datasets.

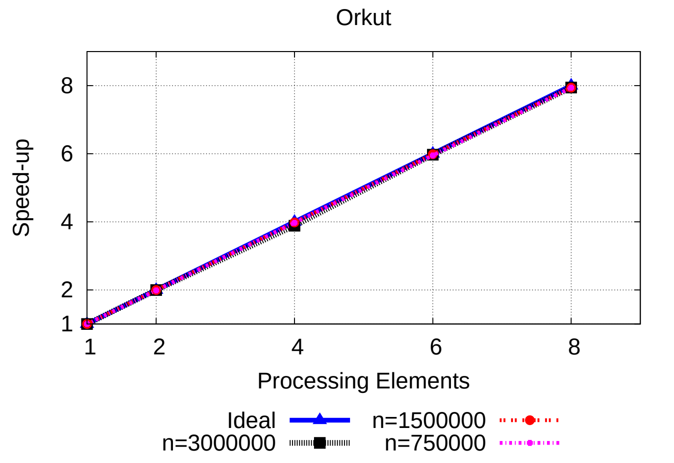


Fig. 6: Comparison of Speed-up of Index Sharing Over APT Algorithm for Different Dataset Sizes (n) in Multi-processor Environment

VII. Conclusion and Future Work

We presented a scalable, parallel solution for the $APSS$ problem based on the index sharing technique. The index sharing technique based parallel $APSS$ algorithm achieves ideal strong scaling performance and this performance remains consistent with variations in the dataset sizes. The work presented in this paper demonstrates that $APSS$ can be performed over large datasets in a reasonable time using parallelization.

Acknowledgment

This work is performed as part of the Scientific Data Management Center (http://sdmcenter.lbl.gov) under the Department of Energy's Scientific Discovery through Advanced Computing program (http://www.scidac.org). Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

References

[1] Sarawagi, S., and Kirpal, A. Efficient set joins on similarity predicates. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, Paris, France*, pp. 743–754.

[2] Arasu, A., Ganti, V., and Kaushik, R. Efficient exact set-similarity joins. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases, Seoul, Korea*, VLDB Endowment, pp. 918–929.

[3] Bayardo, R. J., Ma, Y., and Srikant, R. Scaling up all pairs similarity search. In *WWW '07: Proceeding of the 16th international conference on World Wide Web, Banff, Alberta, Canada*, pp. 131–140.

[4] Xiao, C., Wang, W., Lin, X., and Yu, J. X. Efficient similarity joins for near duplicate detection. In *WWW '08: Proceeding of the 17th international conference on World Wide Web, Beijing, China*, pp. 131–140.

[5] Xiao, C., Wang, W., and Lin, X. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In *Proc. VLDB Endow.*, vol. 1, VLDB Endowment, pp. 933–944.

[6] Awekar, A., and Samatova, N. F. Fast matching for all pairs similarity search. In *WI-IAT '09: IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Milan, Italy.*, pp. 295–300.

[7] Geer, D. Industry trends: Chip makers turn to multicore processors. *Computer 38*, 5 (2005), pp. 11–13.

[8] Agrawal, R., and Shafer, J. C. Parallel mining of association rules. *IEEE Trans. on Knowl. and Data Eng. 8*, 6 (1996), pp. 962–969.

[9] Olman, V., Mao, F., Wu, H., and Xu, Y. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE/ACM Trans. Comput. Biol. Bioinformatics 6*, 2 (2009), pp. 344–352.

[10] Barua, S., and Alhajj, R. High performance computing for spatial outliers detection using parallel wavelet transform. *Intell. Data Anal. 11*, 6 (2007), pp. 707–730.

[11] Dean, J. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining, New York, NY, USA*, pp. 1–1.

[12] Kumar, R., Novak, J., and Tomkins, A. Structure and evolution of online social networks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, Philadelphia, PA, USA*, pp. 611–617.

[13] Mislove, A., Koppula, H. S., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. Growth of the flickr social network. In *WOSP '08: Proceedings of the first workshop on Online social networks* (New York, NY, USA, 2008), ACM, pp. 25–30.

[14] Cheng, X., Dale, C., and Liu, J. Statistics and social network of youtube videos. In *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pp. 229–238.

[15] Breimyer, P., Kora, G., Hendrix, W., and Samatova, N. F. pr: Lightweight, easy-to-use middleware to plugin parallel analytical computing with r. In *IKE '09: Proceedings of the International Conference on Information and Knowledge Engineering, Las Vegas, Nevada, USA* (2009), pp. 667-673.

[16] Charikar, M. S. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, Montreal, Quebec, Canada* (2002), pp. 380–388.

[17] Fagin, R., Kumar, R., and Sivakumar, D. Efficient similarity search and classification via rank aggregation. In *SIGMOD '03: Proceedings of the ACM SIGMOD international conference on Management of data, San Diego, California* (2003), pp. 301–312.

[18] Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. Syntactic clustering of the web. *Comput. Netw. ISDN Syst. 29*, 8-13 (1997), 1157–1166.

[19] Awekar, A., Samatova, N. F., and Breimyer, P. Incremental all pairs similarity search for varying similarity thresholds. In *SNAKDD '09: Workshop on Social Network Mining and Analysis Held in Conjunction with KDD '09, Paris, France* (2009), ACM.

[20] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. Measurement and analysis of online social networks. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, San Diego, California, USA* (2007), pp. 29–42.

[21] Lewis, B., and Berg, D. J. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[22] Google dense hash map library : `code.google.com/p/google-sparsehash/`.

[23] Code and data sets for our algorithms : `www4.ncsu.edu/~acawekar/ike10/`.