Knowledge and Cache Conscious Algorithm Design and Systems Support for Data Mining Algorithms[*]

Amol Ghoting[1][†], Gregory Buehrer[1], Matthew Goyder[1], Shirish Tatikonda[1],
Xi Zhang[1], Srinivasan Parthasarathy[12], Tahsin Kurc[2], and Joel Saltz[12]
[1]Department of Computer Science and Engineering [2]Department of Biomedical Informatics
The Ohio State University, Columbus, OH
PI Contact email: srini@cse.ohio-state.edu

Abstract

The knowledge discovery process is interactive in nature and therefore minimizing query response time is imperative. The compute and memory intensive nature of data mining algorithms makes this task challenging. We propose to improve the performance of data mining algorithms by re-architecting algorithms and designing effective systems support. From the view point of re-architecting algorithms, knowledge-conscious and cache-conscious design strategies are presented. Knowledge-conscious algorithm designs try and re-use repeated computation between iterations and across executions of a data mining algorithm. Cache-conscious algorithm designs on the other hand reduce execution time by maximizing data locality and reuse. The design of systems support that allows a variety of data mining algorithms to leverage knowledge-caching and cache-conscious placement with minimal implementation efforts is also presented.

1. Introduction

The knowledge discovery process is interactive in nature. In fact, interactivity is key to facilitating effective data understanding and knowledge discovery. Typically, during the mining process, the user proceeds in a trial-and-error fashion until the desired results are obtained. In such an environment, minimizing response-time is imperative, because a lengthy delay between responses to two consecutive user queries can disrupt the flow of human perception and the formation of insight. The compute and memory intensive nature of data mining algorithms makes this task especially challenging. To address the aforementioned challenge, the past few years have seen researchers make significant progress in reducing the computational complexity of data mining algorithms. However, the process continues to be time consuming.

We believe that in order to derive high performance on next generation computing infrastructures, one must consider both re-architecting algorithms and designing effective middleware support. From the view point of re-architecting algorithms, we propose to employ two new algorithm design philosophies, namely knowledge-conscious and cache-conscious algorithm designs. Given the iterative and interactive nature of the knowledge discovery process, one expects there to be significant repeated computation through successive executions of a data mining algorithm. A knowledge-conscious algorithm design philosophy attempts to expose this repeated computation, cache it in a knowledge cache, and re-use it during successive executions of a data mining algorithm. It is well known that programs exhibiting poor data locality tend to keep a processor stalled, waiting on the completion of data access for a large fraction of the time. This is also known as the memory wall problem and results in poor CPU utilization. Cache-conscious algorithm designs attempt to alleviate this problem by improving data locality. Furthermore, such designs also allow one to effectively utilize architectural innovations such as chip multiprocessing. From the view point of designing middleware support, one must investigate the types of services that are desired by a range of data mining algorithms and how they can be designed to maximally utilize the available infrastructures.

In this paper we present several mining solutions that leverage the aforementioned views. Specifically, we make the following contributions. First, we present the design of systems support for knowledge-caching and memory placement that can be utilized by a variety of data mining algorithm. Second, we present knowledge-conscious algorith-

[*]This work is primarily supported by NSF grant #NGS-CNS-0406386. The authors would also like to acknowledge NSF grants #CAREER-IIS-0347662 and #RI-CNS-0403342.

[†]Supported by an IBM Ph.D. Fellowship

mic improvements for data clustering algorithms. Third, we present cache-conscious designs in the graph mining and tree mining domains. Fourth, we present an evaluation of our designs.

2. Systems Support

Knowledge Caching: A knowledge-conscious mining framework consists of a *client* and a *server*. The server maintains a *database*, while the client manages a *query queue*, a *query execution engine*, and a *knowledge cache*. The query execution engine accepts a query from the query queue, and executes the query using the contents of the (local) knowledge cache and the (remote) database. Furthermore, using the information gathered through an execution, the query execution engine updates the contents of the knowledge cache to improve performance when answering future queries.

We propose the development of a *data-mining technique centric knowledge cache*, containing recently constructed and/or used knowledge *objects* (Figure 1). Our goal is to build an infrastructure that allows a variety of data mining algorithms to gain the benefits of knowledge caching with minimal implementation efforts. This knowledge cache will be deployed on a cluster of nodes and leverage the combined aggregate main memory and disk space available on such a system.

Data mining algorithms will interface with such a knowledge cache at the abstraction of a user-defined *Knowledge Object*. This *Knowledge Object* must have *Metadata* and *Knowledge* fields within it, as is dictated by the interface. The metadata for a cached knowledge object maintains the following information: user/process identifier, technique adopted (association rule mining), technique-specific parameters (e.g. minimum-support value), and data-specific information. This metadata can be used to determine if a cached knowledge object can be reused for a query. While *Metadata* field encodes information about the information stored in the knowledge object, the *Knowledge* field encodes the actual knowledge. Data mining applications often rely on specialized pointer-based data structures, for storing the results of queries to facilitate interactive exploration. To deal with this issue, the user must implement *linearize* and *delinearize* methods for the *Metadata* and *Knowledge* fields. The *linearize* method allows the system to convert the *Metadata* and *Knowledge* fields into a binary block of memory, allowing for transmission over a network and efficient storage. The *delinearize* method allows the system to convert these fields into an algorithm interpretable form.

The data mining algorithms will interact with the knowledge-cache using *get* and *put* methods. The *put* method accepts a *Knowledge Object* from the client and inserts it into the knowledge cache. The *get* method retrieves a knowledge object from the cache using a user defined *Query*

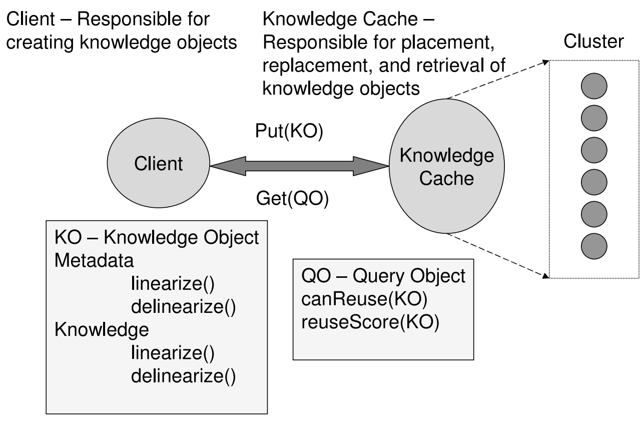

Figure 1. Knowledge Cache

object. This *Query* object associates a re-use score with each knowledge object and the most re-usable *Knowledge Object* is returned by the knowledge cache.

The knowledge cache will be responsible for managing both local disk space and memory space alloted for the cache dispersed across multiple nodes. With a distributed cache, one important issue is the effective management of distributed space and bandwidth. In a heterogeneous environment, memory and bandwidth can vary across the machines in the system. Moreover, the memory space on a machine can be shared between the application and the cache. Thus, the cache space on a machine may shrink, requiring that some of the cached objects be evicted. We propose to investigate adaptive strategies for placement and eviction of cached objects in the system. One approach is to assign a priority to each cached object. This priority can be based on the static and dynamic attributes used by cache replacement policies (e.g., how recently the object has been accessed, the popularity of the object relative to others, the size of the object, and the cost of computing the object). A high priority object will be cached on a machine with high bandwidth. Since a cached object is made up of fixed size cache blocks, we will also develop strategies to distribute the blocks across the system to take advantage of distributed bandwidth and cache space. When a cached object needs to be evicted from a machine, it can be staged to the local persistent cache on the machine or it can be transferred to other machines in the environment. When selecting the machine to transfer the object to, the cache management strategy should take into account the priority of the object as well as the cost of the transfer operation; for instance, a single transfer operation can create a cascade of transfers if the selected machine does not have space to accommodate the object. We will examine strategies to address these issues.

Memory Placement: We have also developed middleware that handles memory placement for complex data structures to improve reference locality for data mining algorithms. The middleware supports different placement policies and does not pollute the cache with boundary tag information.

We illustrate (Figure 2) these policies in the context of the hash-tree structure that is popular in data mining. We employ a localized placement policy that groups related data structures together using local information present in a single subroutine. In this policy, we utilize a reservation mechanism to ensure that a list node (LN) with its associated itemset (Itemset) in the leaves of the final hash tree are together whenever possible, and that the hash tree node (HTN) and itemset list header (ILH) are together. This is done because an access to a list node is always followed by an access to its itemset, and an access to a leaf hash tree node is followed by an access to its itemset list header. We also employ a global placement policy that utilizes the knowledge of the entire hash tree traversal order to place the tree in memory. In our case, the hash tree is remapped in depth-first order, which closely approximates the hash tree traversal.

3. Knowledge-Conscious Clustering

We consider the problem of executing exploratory data clustering queries[1]. Here, the user is interested in interactively clustering different subsets of the data set D. Furthermore, during this process, the user is also interested in varying k, the desired number of clusters. Such an exploration of the data can provide the user with a much deeper understanding of the evolving behavior of the clusters [1].

Given a data set D consisting of n data points, each with d dimensions, the data clustering problem is to partition this data set into k subsets such that each subset behaves "well" under some measure. The popular kMeans clustering algorithm can be briefly described as follows. First, it begins with k random centers, $C^0 = \{C_1^0, \cdots, C_k^0\}$. Next, for each of the n data points, it finds its closest center in C^0. The data points are partitioned into k subsets based on their closest centers. The center of mass for each of these k subsets is used to find the new set of k centers, $C^1 = \{C_1^1, \cdots, C_k^1\}$. This process continues iteratively until we encounter an iteration i such that the centers C^i and $C^{(i+1)}$ are identical. Each iteration of this naive kMeans algorithm scales as $O(nkd)$.

The state-of-the-art kMeans clustering algorithm, due to Pelleg and Moore, improves the performance of the above mentioned algorithm by employing a *multi-resolution kd-tree* [7]. Multi-resolution kd-trees have the following properties. First, they are binary trees. Second, each node in the kd-tree contains information about all points contained in a hyper-rectangle h. This hyper-rectangle is stored at the node using two boundary vectors h^{max} and h^{min}. At each node is also stored the *number* and the *center of mass* of all points that lie within h. All the children of a node represent hyper-rectangles contained within h. Third, each node has a *split dimension* and a *split point* assigned to it. The value of the split point on the split dimension is referred to as the *split value* of the node. The children of a node represent two hyper-rectangles such that all points with values less than the split value on the split dimension are assigned to one child, and all points with values greater than the split value on the split dimension are assigned to the other child. This data structure has exactly n nodes.

Given a set of centers C^i and a hyper-rectangle h, $owner(h, C^i)$ is defined as the center $c \in C^i$ for which any point in h is closer to c than any other center in C^i. Note that h does not always have an owner in C^i. Pelleg and Moore used this concept of ownership to improve the performance of the kMeans algorithm. The multi-resolution kd-tree is used to assign the points to the k centers in each iteration. The algorithm proceeds recursively and can be briefly described as follows. First, beginning with the root node, it checks to see if the hyper-rectangle associated with the node has a unique owner in C^i. If we have a unique owner, statistics stored at the node (*number of points* and *center of mass*) can be used to assign all points covered by the hyper-rectangle to the unique owner, and the procedure can then return. Otherwise, the split point associated with the node is assigned to one of the k centers, and the children of the node are processed in a similar fashion, recursively.

In order to perform exploratory data clustering using the state-of-the-art, a system typically proceeds as follows. First, it queries the remote database to retrieve the desired subset of the data. Next, it builds a multi-resolution kd-tree using the data retrieved from the database. Finally, it uses the aforementioned variant of the kMeans algorithm (due to Pelleg and Moore) to cluster the data.

Existing solutions to exploratory data clustering suffer from the following drawbacks. First, when the database it very large and cannot be cached on the client's side, during query execution, the system needs to retrieve a significant amount of data from the remote database. This is often time-consuming. Second, there is no re-use of computation between iterations of the kMeans algorithm and between executions of two different kMeans clustering queries. This redundant computation is often excessive and significantly affects performance. We propose to facilitate exploratory data clustering by employing a *client-side knowledge cache* to tackle the above mentioned challenges.

Reducing Remote I/O: In order to reduce remote I/O during execution, we propose to maintain a *low-resolution summary* of the data set on the client's side. This low-resolution summary must have the following characteristics. First, given that a summary with a satisfactory resolution is available, a kMeans clustering using this summary should be identical to that when using the entire data set. Second, when the clustering cannot be performed accurately, we should be able to improve the resolution of this summary to the desired level, incrementally, accessing only a small subset of the remote database. After query execution, for each kd-tree that is built, the sub-tree that is accessed when executing the kMeans query is stored at the client. This *cached sub-tree* is a low-resolution summary of the data in a block and can be used to answer future queries. Furthermore, depth-first data re-ordering allows the cached sub-tree to be grown incrementally.

Reducing the Number of Candidate Centers between Iterations: When executing a kMeans query, the main operation is that of assigning a point or a hyper-rectangle to one of the k candidate centers. In order to make an assignment, for a data point, we need $O(k)$ computations, and for a hyper-rectangle, we need $O(k^2)$ computations. Using the execution history of a query, we propose to reduce the set of candidate centers, and thereby reduce the number of computations needed to make an assignment.

Let us assume that in iteration i of a kMeans query, data points and hyper-rectangles in the cached sub-tree are assigned to one of k centers in C^i. Let $C^{(i+1)}$ be the new set of k centers to be used in $(i+1)^{th}$ iteration. Let $rad(C^i_j)$ be the radius of the j^{th} center in C^i. Let $d(C^i_j, C^i_k)$ be the euclidean distance between centers C^i_j and C^i_k. Let $maxdist(C^i_j, h)$ be the maximum distance between h (a hyper-rectangle or a point) and a center C^i_j.

Lemma 1 *If a hyper-rectangle or a data point is assigned to center C^i_j in iteration i, then in the $(i+1)^{th}$ iteration, it cannot be assigned to any center $C^{(i+1)}_k$ for which $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k) < d(C^i_j, C^i_k)/2 - rad(C^i_j)$.*

Lemma 2 *If a hyper-rectangle (or a data point) h is assigned to center C^i_j in iteration i, then in the $(i+1)^{th}$ iteration, it cannot be assigned to any center $C^{(i+1)}_k$ for which $d(C^i_j, C^{(i+1)}_j) + d(C^i_k, C^{(i+1)}_k) < d(C^i_j, C^i_k)/2 - maxdist(C^i_j, h)$.*

The kMeans using kd-trees algorithm is very easily modified to benefit from the aforementioned lemmas [5]. These lemmas can also be used to reduce redundant computations between multiple kMeans queries.

A scaling factor of 0.05 represents a data set with well separated clusters. Figure 3 shows the time required for remote I/O and computation for the naive kMeans, the kMeans using kd-trees, and the knowledge-conscious kMeans algorithms. We see up to a 10-fold reduction in remote I/O time and 6-fold reduction in computation due to knowledge re-use for the knowledge-conscious kMeans algorithm, an overall 8-fold reduction in execution time.

4. Cache-conscious Algorithm Designs

Over the past decade, processor speeds have increased much faster than DRAM speeds, leading to the creation of a gap between processor performance and memory system performance. Programs that exhibit poor data locality leave the processor stalled for a large fraction of the execution time due to long memory latencies. In such situations,

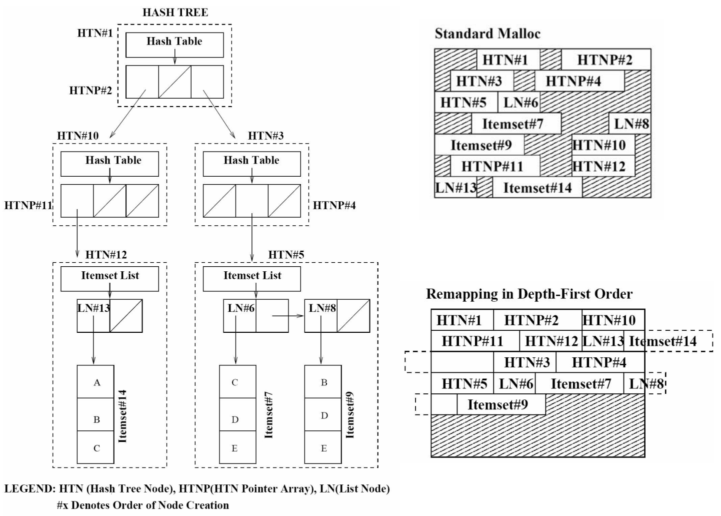

Figure 2. Memory placement

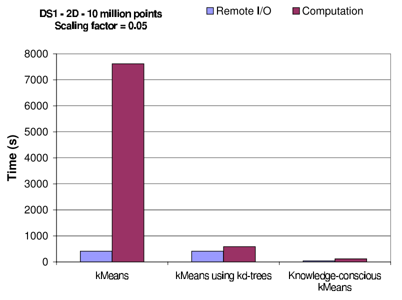

Figure 3. Reduction in remote I/O and computation - DS1

maximizing data locality is imperative. Furthermore, it is commonly accepted that Chip Multiprocessors (CMPs) will become the *de facto* standard for commodity PCs. These processors contain multiple processing cores on the same die. The cores not only share all off-chip resources, but many on-chip resources, such as cache, making memory bandwidth a bottleneck. Therefore, *it is paramount that algorithm designers research effective strategies to incorporate task parallelism and efficient memory system usage to ensure that application performance is commensurate with such architectural advancements*[3, 4, 6].

Graph Mining: Finding frequent patterns in graph databases such as chemical and biological data, XML documents, web link data, financial transaction data, social network data, and other areas has posed an important challenge to the data mining community. A major challenge in substructure mining (also termed graph mining) is that the search space is exponential with respect to the data set, forcing runtimes to be quite long. These long runtimes can be mitigated by parallelizing the workload, and executing it on a CMP.

To achieve high scalability in a parallel graph mining algorithm, one must overcome several fundamental obstacles. The two most challenging of these are load imbalance and efficient utilization of the memory hierarchy. The basis for

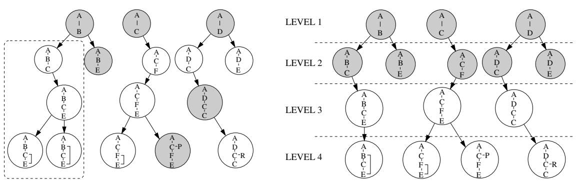

Figure 4. Adaptive vs. levelwise partitioning.

effective load balancing is a task partitioning mechanism possessing sufficient granularity so as to allow each processing element to continue to perform useful work until the mining process is complete. In itemset mining, for large databases frequent-1 items generally suffice. However, for graph mining this is not the case. In graph mining, a single frequent-1 item may contain 50% or more of the total execution cost. This is because task length is largely dependent on the associativity in the dataset.

CMP systems typically have far less cache per processor than other parallel architectures, including message-passing clusters or shared memory multiprocessors. Sharing of the L2 cache is of particular concern. If multiple threads have independent working sets, then the effective size of the cache is significantly diminished. This is predicated by the real estate constraints of the chip, since for a fixed chip size each additional core will use silicon previously dedicated to cache. In graph mining, data accesses often lack temporal locality because the projected data set is typically quite large. These accesses also lack good spatial locality because most of the data structures employed are pointer-based.

We performed a simple working set study to evaluate the benefits of improved dataset graph accesses. We compared two strategies for parallel graph mining. First, we forced threads to always mine their own child tasks. Second, we allowed threads to mine other threads' tasks; that is, task stealing. The cache miss rates for the first strategy were, on average, three times lower than the second strategy. The reason is that when a task is stolen, typically most of the needed data set graphs are not in cache. As such, *algorithm designers should explore and optimize the tradeoffs between improved temporal locality and proper load balancing*.

Our solution to maintain excellent cache utilization while still affording load balance is called *Adaptive Task Partitioning*, and is shown in Figure 4. Every mining task generates some number of child candidates from which to extend the currently mined graph. Each candidate is then recursed upon. With *adaptive partitioning* [2], each processor makes a decision at runtime for each frequent child extension. A child is an edge tuple *(srcNode,destNode,srcLabel,edgeLabel,destLabel)* which can be added to the currently mined graph to create a new frequent subgraph. The child task may be mined by the creating processor, or enqueued.

The decision whether to mine the task is based on the current load of the system. We do not require a specific mechanism to make this decision, because in part it is based on the queuing mechanism used. In our experiments, we evaluate the size of the local thread's queue, using a threshold of 10% of the total number of frequent-1 edges. Another option could be to set a threshold for the smallest current size of any queue.

At each step in the depth-first mining process, each subtask can be enqueued into the system for any other processor to mine. Thus the granularity of a task can be as small as a single mining call, which is rather fine-grained. Adaptive partitioning is depicted in Figure 4 (left). In this example, task (A-B) had two children, namely (A-B-C) and (A-B-E). Tasks which were enqueued are shaded gray. A circle has been drawn around a task which was allowed to grow dynamically. Task (A-B-E) was enqueued, while task (A-B-C) was mined by the parent process. After mining (A-B-C), only one child was created, which was kept by the parent. The subsequent two children were both mined by the parent, because the queues had sufficient work so as to allow it.

The performance difference between full partitioning and adaptive partitioning is primarily due to the poor cache performance exhibited in full partitioning. We perform a working set study to compare the temporal locality of the two strategies in the context of graph mining. We use Cachegrind on a single processor machine (Pentium 4 machine). Because Cachegrind currently does not profile multiple threads, we simulate 32 threads by allowing a single thread to remove from any point within 32 locations from the head of the queue with equal probability. As seen from the results in Figure 5, *adaptive partitioning reduces the miss rates by 50%.* This is due to the improved temporal locality. In adaptive partitioning, there is a much higher probability that the processor which mines a parent task will also mine its child tasks. Child patterns are extensions of parent patterns, thus the database objects which embed a child must be a subset of the database objects which embed its parent. As such, the database objects which embed the child task's pattern are more likely to be in cache. The end result is improved scalability.

Tree Mining: We proposed new algorithms for mining frequent subtrees from a database of trees [8]. Researchers have shown frequent tree mining to be useful in wide variety of applications. It is first applied in bioinformatics in order to find the similarity of phylogenetic trees, and RNA structures. Web usage analysis can benefit from frequent tree mining as the user's access patterns of a website are often modeled as trees. It is also shown to help in XML query selectivity estimation, designing XML classifiers, and network multicast routing algorithms. We designed two new algorithms, namely TRIPS and TIDES to mine rooted subtrees which can either be induced, embedded, labeled, unlabeled, ordered, unordered, or edge-labeled. These generic

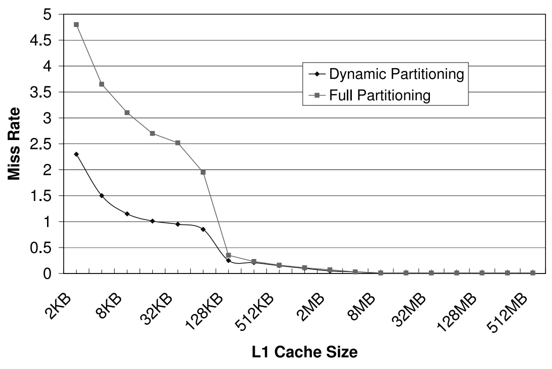

Figure 5. Working sets for adaptive partitioning and full partitioning.

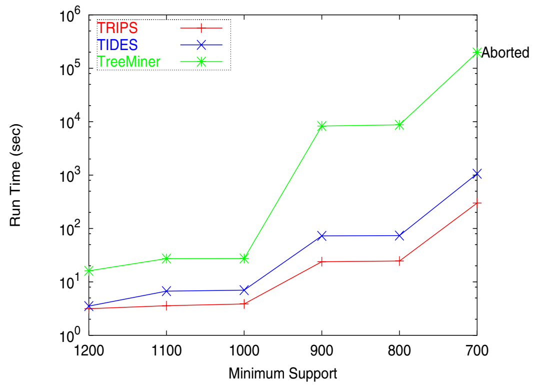

Figure 6. Performance of TRIPS and TIDES on Cslogs

algorithms operate on the sequence based representation of the database trees to efficiently generate and evaluate the candidate subtrees. TRIPS and TIDES differ in their use of tree-sequencing methods. TRIPS uses the Prufer sequence encoding whereas TIDES is based on depth-first sequences. Similar to other pattern mining algorithms, our algorithms also have two important steps, candidate generation and support counting. The candidate generation process that is transformed into a relatively easy process of sequence extension is *complete* and *non-redundant*. It generates every possible subtree in the search space and generates each subtree only once. Our novel candidate generation process also makes the support counting step a trivial task of accumulating the counts. The algorithms make use of a meta-structure, embedding list, to speedup the mining process. Since both the data representation and the meta-structures are based on simple arrays, our algorithms exhibits excellent cache performance.

Figure 6 shows the performance of our algorithm on the Cslogs data set compared to a state-of-the-art algorithm TreeMiner. Clearly, our algorithms out performs TreeMiner achieving a speedup up to 355 times on Cslogs. This efficiency is achieved while using very less main memory when compared to TreeMiner. For example, on Cslogs data set, at minimum support of 800, TreeMiner used 2.4GB of main memory (RSS size) whereas our algorithms used only 60MB. Added to that, our array based data structures for both storing data and meta information make our algorithms cache-friendly. In fact, on tested real and synthetic data sets, our algorithms achieved average L1, L2 hit rates of 98%, 99%, respectively and an average CPI of 0.7. More detailed results are available in [8].

5. Conclusions

In this paper, we presented knowledge-conscious and cache-conscious design strategies to improve the performance of data mining algorithms that are both compute and memory intensive. Knowledge-conscious designs target the computation that is repeated between iterations and executions of a data mining algorithm. Cache-conscious designs improve performance by targeting data locality. First, the design of systems support that can ease the development of knowledge-conscious solutions and handle memory placement, for a variety of data mining algorithms, is presented. Second, we showed how data clustering algorithms can be made knowledge conscious. Finally, we demonstrated that cache-conscious designs in the graph mining and tree mining domains can provide significant performance improvements.

References

[1] C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for clustering evolving data streams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2003.

[2] G. Buehrer, S. Parthasarathy, and Y. Chen. Adaptive parallel graph mining for cmp architectures. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2006.

[3] G. Buehrer, S. Parthasarathy, and A. Ghoting. Out-of-core frequent pattern mining on a commodity pc. In *Proceedings of the International Conference on Knowledge Discovery and Data mining (KDD)*, 2006.

[4] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Towards terabyte data mining: An architecture conscious solution. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2007.

[5] A. Ghoting and S. Parthasarathy. Knowledge-conscious exploratory data clustering. In *Proceedings of the International Conference on Principles of Data Mining and Knowledge Discovery*, 2006.

[6] A. Ghoting, S. Parthasarathy, and M. Otey. Fast mining of distance-based outliers in high dimensional datasets. In *Proceedings of the SIAM Conference on Data Mining*, 2006.

[7] D. Pelleg and A. Moore. Accelerating exact kmeans algorithms with geometric reasoning. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1999.

[8] S. Tatikonda, S. Parthasarathy, and T. Kurc. Trips and tides: new algorithms for tree mining. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 455–464. ACM Press, 2006.