Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution

Jiayuan Meng[†‡], Anand Raghunathan[†§], Srimat Chakradhar[†], and Surendra Byna[†]
[†] NEC Laboratories America, Princeton, NJ
[‡] Department of Computer Science, University of Virginia, Charlottesville, VA
[§] School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN

***Abstract*—It is widely believed that most Recognition and Mining (RM) workloads can easily take advantage of parallel computing platforms because these workloads are data-parallel. Contrary to this popular belief, we present RM workloads for which conventional parallel implementations scale poorly on multi-core platforms. We identify off-chip memory transfers and overheads in the parallel runtime library as the primary bottlenecks that limit speedups to be well below the ideal linear speedup expected for data-parallel workloads. To achieve improved parallel scalability, we identify and exploit several interesting properties of RM workloads — sparsity of model updates, low spatial locality among model updates, presence of insignificant computations, and the inherently self-healing nature of these algorithms in the presence of errors. We leverage these domain-specific characteristics to improve parallel scalability in two major ways. First, we utilize data dependency relaxation to simultaneously execute multiple training iterations in parallel, thereby increasing the granularity of the parallel tasks and significantly lowering the run-time overheads of fine-grained threading. Second, we strategically drop selected computations that are insignificant to the accuracy of the final result, but account for a disproportionately large amount of off-chip (memory and coherence) traffic.**

Through the application of the proposed techniques, we show that much higher speedups are possible on multi-core platforms for two important RM applications — document search using semantic indexing, and eye detection in images using generalized learning vector quantization. On an 8-core platform, we achieve application speedups of 5.5X and 7.3X compared to sequential implementations. Compared to conventional parallel implementations of these applications using Intel's TBB, the proposed techniques result in 4.3X and 4.9X improvements. Although the optimized parallel implementations are not numerically equivalent to the sequential implementations, the output quality is shown to be comparable (and within the margin of variation produced by processing the input data in a different order). We also explore error mitigation techniques that can be used to ensure that the accuracy of results is not compromised.

***Keywords*-Best-effort computing; Parallel computing; Parallel programming; Recognition; Mining; Multi-core; Dependency relaxation.**

I. Introduction

Recognition and Mining (RM) represent an emerging class of computing workloads that are expected to be prevalent on future multi-core and many-core computing platforms. RM applications address the digital data explosion problem by building mathematical or statistical models that help us to interpret and understand raw data, and use these models to search through massive amounts of data [8], [12]. Several studies have demonstrated that many RM algorithms are data parallel workloads and can easily take advantage of parallel computing platforms [9], [22]. In other words, their performance is expected to scale well on future multi-core and many-core platforms. While this claim is true for a good number of RM workloads, we demonstrate that not all RM workloads fit this stereotype. In other words, some important RM applications contain significant parallelism that cannot be profitably exploited on current multi-core platforms using conventional parallel implementations. We present and analyze two representative applications that fall into this category: document search based on Supervised Semantic Indexing (SSI), and eye detection in images based on Generalized Learning Vector Quantization (GLVQ). We address the challenge of improving the performance of these applications on multi-core platforms. Towards that end, we propose a domain-specific parallelization technique that leverages the computational structure and domain characteristics of RM applications.

In this work, we specifically consider RM algorithms that follow a computational template wherein a model or intermediate result is iteratively refined based on the provided input training or testing data. The reads and writes to the model introduce data dependencies between successive "iterations". In conventional parallel implementations, iterations are therefore executed serially, and parallelism within each iteration is exploited. This approach, however, may limit the algorithms' parallel scalability due to a variety of factors such as the granularity of parallelism, or communication and synchronization between parallel tasks. To address this issue, we propose an alternative approach to parallelizing RM algorithms, which creates and exploits parallelism across iterations by judiciously relaxing or ignoring some of the data dependencies between iterations. Our approach goes beyond conventional parallel implementation techniques because we focus on obtaining comparable output quality rather than imposing the restriction of maintaining numerical equivalence to the sequential implementation. The following insights into the target applications allow us to take this approach:

- Each iteration is likely to update only a small, data-dependent portion of the model. We refer to this as the sparsity of model updates.
- Successive iterations are highly likely to update different parts of the model. In other words, model modifications display minimal locality across iterations.
- Occasional errors introduced in the model by executing iterations in parallel while relaxing data dependencies can be tolerated by the inherently resilient (statistical and self-correcting) nature of RM algorithms, causing either imperceptible or acceptable variation in the final output.

From a more general perspective, these properties attribute to the forgiving nature of RM applications: they accept input data that is noisy and redundant, they perform computations that are statistical in nature, and they can produce a large number of (numerically) distinct solutions that are all considered equivalent or acceptable. In the context of parallel computing, this forgiving nature implies that *numerical equivalence of results between parallel and sequential implementations is not mandatory*; different implementations are equivalent as long as they produce results of similar quality (classification accuracy, detection probability, relevance of search results, *etc.*).

Figure 1 qualitatively demonstrates the net effect of the proposed techniques, which is to improve application performance and scalability with increasing numbers of cores. This is achived in two ways. First, dependency relaxation allows coarse-grained parallelism with minimal communication or synchronization. Its performance can scale to a larger number of cores until other limits such as the

978-1-4244-6443-2/10$26.00©2010 IEEE

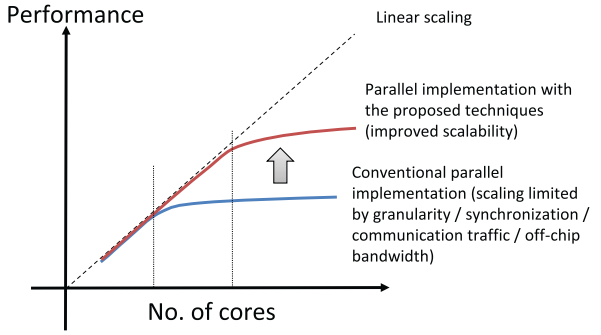


Figure 1. Conceptual illustration of impact of data dependency relaxation on performance and parallel scalability of RM applications

off-chip bandwidth bottleneck set in due to memory and coherence traffic. We demonstrate that the impact of off-chip bandwidth on performance scaling can be alleviated by once again exploiting the forgiving nature of RM applications. We identify a small set of computations that have minimal impact on the output, but cause a disproportionate amount of memory or coherence traffic. These computations, together with the associated memory accesses, are dropped, resulting in improved parallel scalability.

In our previous work [25], we proposed the concept of best-effort parallel computing and introduced a parallel programming model that naturally emobides this concept. In this paper, we study RM applications' parallel scaling bottlenecks in detail and apply best-effort techniques to address these bottlenecks. Specifically, this paper makes the following contributions:

- The bottlenecks in scaling two significant RM applications are investigated quantitatively using state-of-the-art profiling tools.
- The forgiving nature of these applications is characterized to justify the proposed techniques.
- Data dependency relaxation is suggested as a tool to mitigate runtime library overheads.
- Memory and coherence traffic is reduced to address the bandwidth bottleneck by leveraging the forgiving nature to drop non-critical memory operations and their corresponding computations.
- New techniques are proposed to mitigate the loss of output quality due to the use of best-effort techniques.

We apply our techniques to the SSI and GLVQ applications and evaluate the improvements in performance and scalability on an 8-core Dell Poweredge 2950 server platform. Overall, we achieved application speedups of upto 5.5X and 7.3X respectively over sequential implementations. These correspond to speedups of 4.3X and 4.9X respectively over conventional parallel implementations using Intel's Threading Building Blocks (TBB) [21]. Although our parallel implementations do not preserve numerical equivalence with the sequential implementation, we argue that this is perfectly acceptable for RM applications, where there is often no "golden" result (*e.g.*, providing input data in a different order leads to a different result even for the sequential implementation). We demonstrate that our optimized parallel implementations produce results of comparable quality (the output falls within a small, tolerable range). In addition, we propose techniques that can be used to mitigate any loss of quality, and demonstrate that identical accuracy can be achieved while maintaining significant performance improvements.

The rest of the paper is organized as follows. In Section II, we describe the applications used in our work, and the underlying algorithms. In Section III, we motivate our work by analyzing conventional parallel implementations of these applications on an 8-core platform and demonstrating their scaling bottlenecks. In Section IV, we analyze the relevant characteristics of RM applications to make a case for data dependency relaxed parallel implementations, supporting our observations through empirical data. In Section V, we present data dependency relaxed versions of the SSI and GLVQ algorithms, and examine the impact of the proposed technique on application performance and output quality. In Section VI, we present techniques to address the off-chip bandwidth bottleneck by dropping selected computations that have insignificant impact on the output but cause disproportionately high memory or coherence traffic. In Section VII, we present techniques to mitigate the impact of data dependency relaxation on applications' output quality. We present related work in Section VIII, and conclusions in Section IX.

II. Background

In this section, we introduce two representative RM applications that are used as drivers for this work. Specifically, we consider document search using Supervised Semantic Indexing (SSI) [2], and eye detection using Generalized Learning Vector Quantization (GLVQ) [26], [30], [28]. For both applications, training or building the underlying model is much more computation-intensive than applying the learned model; therefore, we focus on the training phase. As shown in the next section, these applications represent RM workloads that are not easily scalable in performance on multi-core platforms. While we use these two applications for our demonstration, we believe that our observations and the proposed techniques apply to other RM workloads with similar characteristics, as discussed further in Section V.

A. Supervised Semantic Indexing

Supervised Semantic Indexing [2] is an algorithm used for ranking documents in a corpus or database based on their semantic similarity to a given text-based query. It performs this by establishing a direct association between the word content of a document and its similarity score, taking into account correlations between words due to synonymy and polysemy. It has demonstrated state-of-the-art performance in accuracy on large data sets such as Wikipedia [2].

In SSI, each document is represented by a vector of TFIDFs for each word in the vocabulary [1]. The TFIDF is a product of Term Frequency (TF) and Inverse Document Frequency (IDF). TF measures the importance of a word in a document. It is the ratio of the number of occurrences of a word in a document to the total number of occurrences of all words in that document. IDF is a measure of the general importance of a word in all documents. The IDF of word is obtained by dividing the number of all documents by the number of documents containing the word, and then taking the logarithm of that quotient [1]. Using the trained model, this long TFIDF vector can be transformed to a much shorter *semantic vector*, representing the likelihood of a document in a number of machine-learned conceptual categories. The dot product between two semantic vectors measures the semantic similarity between their corresponding documents. In the context of document search, documents are ranked and retrieved based on their similarity with the query document.

The SSI model is learned similar to an artificial neural network (ANN) [29]; the input nodes are the TFIDFs of all words in the vocabulary, whose values correspond to

[1]Since each document is likely to contain only a small fraction of the vocabulary, this vector is represented as a sparse vector.

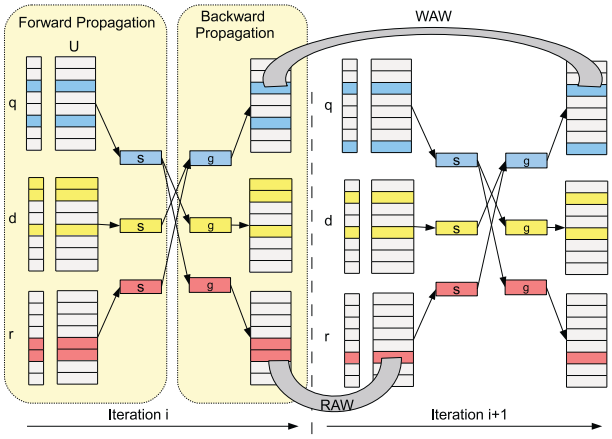


Figure 2. The computation structure of SSI training and its dependency analysis. Colored elements in q, d, and r represent words with nonzero TFIDFs. In every iteration, only those colored rows in U that correspond to nonzero TFIDFs are used and updated.

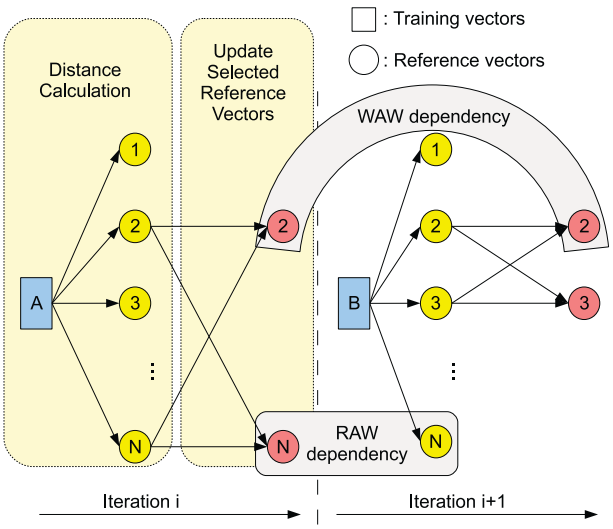


Figure 3. The computation structure of GLVQ training and its dependency analysis.

an input document. Each input node is connected directly to all output nodes that represent a predefined number of conceptual categories to be learned. The ANN can be represented as a bias vector b with a length of C and a weight matrix U sized $N \times C$, where N is the number of words in the vocabulary and C is the predefined number of conceptual categories. The i^{th} column of U stores each word's weight in contributing to the likelihood of a document belonging to the i^{th} conceptual category. This model is refined in iterations, where each training iteration involves three TFIDF vectors representing a query document (q), a labeled relevant document (d), and an irrelevant document (r). Figure 2 illustrates the computations involved in two consecutive iterations of the SSI training algorithm. In each iteration, the model is updated by performing the following steps:

1) Forward propagation: each of the three documents (q, d, and r) is fed forward through the network independently, producing three semantic vectors, each of which is calculated as weighted sums of TFIDFs for each conceptual category ($s_q = q^T U + b^T$, $s_d = d^T U + b^T$, and $s_r = r^T U + b^T$).
2) Comparing relevancy: The relevance scores of d and r to q are calculated as $s_q \cdot s_d$ and $s_q \cdot s_r$, respectively, where "$\cdot$" denotes the dot product. If the score of d is not greater than that of r by a specified threshold, three feedback gradients vectors are set to nonzero ($g_q = s_d - s_r$, $g_d = s_q$, and $g_r = -s_q$). These gradient vectors are then fed to the backward propagation step.
3) Backward propagation: it is only computed when the gradient vectors are nonzero. It updates U and b using q, d, and r, and the gradients. Each document's TFIDF vector is multiplied by their corresponding gradient vector to generate a gradient matrix, which is scaled according to a predefined learning rate ξ and added to U using the equation $U = U + \xi \times (q \times g_q^T + d \times g_d^T + r \times g_r^T)$. These gradient vectors are also scaled and added to b using the equation $b = b + \xi \times (g_q + g_d + g_r)$.

In our experiments, we use Wikipedia's corpus [17] composed of 1,863,574 documents to train the model. The number of words in the vocabulary (N) is 30,000 and the number of conceptual categories (C) is selected as 200 in order to provide sufficient modeling capacity while avoiding over-fitting [11]. For each document, its relevant documents are automatically labeled according to the links provided by Wikipedia (a document is relevant to all the other documents to which it has links). The relevant document, r, is randomly picked from the corpus. The TFIDF vector for each document is computed in a pre-processing step before the training process. While N is large and thus U is large as well, not all rows of U are accessed in each iteration due to the sparsity of words in q, d and r. In fact, of the 30,000 rows in U, only those that correspond to nonzero components in q, d and r are used to compute the semantic vectors in the forward propagation. Moreover, it is exactly the same rows in U that are updated in the backward propagation, should the relevancy comparison determine that backward propagation is necessary. Training is organized into epochs — in each epoch, U is trained by 10,000 tuples of q, d and r. At the end of each epoch, error is measured using another set of 10,000 tuples of q, d and r. A testing instance is regarded as successful if $s_q \cdot s_d > s_q \cdot s_r$. Accuracy is calculated as the number of successful instances divided by the total number of testing instances. Tuples used for testing are different from those used for training.

B. Eye Detection Using GLVQ

GLVQ [26], [30], [28] is a supervised learning algorithm that classifies an input vector into one of a pre-specified number of categories or classes. It operates by measuring the distances from the input vector to a set of trained *reference vectors*. Each reference vector belongs to one of the classes, while each class may have multiple reference vectors associated with it. Given a set of trained reference vectors, classification of new input vectors is performed by just choosing the class that has the closest reference vector.

In the training process, reference vectors are learned using labeled training vectors. Figure 3 illustrates the computations involved in two consecutive iterations of the GLVQ training algorithm. In each iteration, a training vector is processed to update reference vectors in three steps:

1) The distances between the training vector and all reference vectors are calculated.
2) Two reference vectors are picked for update: one is the closest reference vector R1 in the same class as the labeled training vector; the other is the closest reference vector R2 among the reference vectors of all other classes.

3) R1 and R2 are updated such that R1 moves closer to the training vector, and R2 moves further away from it.

In the application of the GLVQ algorithm to eye detection, we are given an image segment (typically cropped from a larger image), and need to determine whether it contains an eye or not. Training and Testing vectors are computed from the given images as histograms of gradients, where each vector has 512 dimensions. The two categories, eye and non-eye, have 64 reference vectors each. In the training process, pre-labeled eye and non-eye images are used to train the reference vectors. The reference vectors are initialized to random training vectors in their corresponding category. The training vectors are fed to the GLVQ training algorithm in a randomized order. Once the training process is completed, the model (reference vectors) may be used for classification. Classification accuracy is calculated by dividing the correctly classified image segments by the total number of testing image segments.

III. Motivation

In this section, we analyze the performance of the SSI and GLVQ applications on an 8-core platform and show that they exhibit poor performance scalability, motivating the domain-specific parallelization technique proposed in this work. While we use these two applications for demonstration, the same principle applies to other RM workloads with similar characteristics, as discussed further in Section V.

A. Parallel Scalability of SSI

We analyze the computational structure of the SSI training algorithm to identify opportunities for parallelization, and then discuss the measured performance of a conventional parallel implementation on an 8-core platform.

As Figure 2 illustrates, consecutive iterations may have RAW and WAW dependencies if they select two sets of documents that share some common words — in such cases, the previous iteration *may* update some rows of U that will be used in the next iteration for calculating the semantic vectors, and these rows may be modified again. Therefore, training iterations have to be computed sequentially. It may appear that there is still sufficient parallelism within the most time-consuming phases (*i.e.*, forward propagation and back propagation in each training instance). However, as shown next, this parallelism does not translate to significant performance improvement on an 8-core platform.

We developed a parallel implementation of the SSI application using Intel's TBB, and evaluated it on the Dell Poweredge 2950 8-core platform. We explored different ways of parallelizing the SSI training algorithm. Both forward propagation and backward propagation can be parallelized. At a higher level, the computation of s_q, s_d, and s_r during forward propagation can be performed in parallel. In addition, the calculation of each semantic vector can be parallelized across different rows of U. The backward propagation for each feedback gradient vector can be parallelized across different rows of U as well. However, this conventional parallel implementation results in poor scaling, as shown in Figure 4(a) (only 1.3X speedup is achieved using 8 threads).

In order to determine the cause of the poor performance and scalability, we used Intel's VTune performance analyzer [32] to profile the training process as it executed on the 8-core platform. We measured the breakdown of processing cycles between the application and the run-time overheads (TBB, the threading library). The results in Figure 4(b) shows that the proportion of time spent in run-time overheads increase from 17% to 50% as the application scales from 1 thread to 8 threads [2]. Besides run-time overheads, SSI's performance also suffers from limited memory bandwidth. Due to the random selection of documents from the large corpus and the large size of the model, the memory requirements are quite high, and only increase with larger numbers of threads. Using VTune, we illustrate in Figure 4(c) that the bus utilization increases beyond 50% with 6 threads [3]. Moreover, a significant portion of the bus bandwidth is consumed by the run-time libraries.

To further validate our conclusion, we changed the number of conceptual categories (to a very small value - 8, and a large value - 800), and measured the performance of the parallel implementation. The results are reported in Figure 4(a). We observe that performance improvement from parallelization is higher when the number of conceptual categories (C) increases, which essentially gives each parallel thread more workload and increases the computation-to-memory-access ratio. However, increasing the number of concepts beyond 200 increases the overall execution time of the algorithm significantly without any noticeable improvement in accuracy. In effect, we are creating a more scalable workload by performing useless computation! We expect scaling of the workload to be primarily in terms of the size of the input data set (number of documents in the corpus), and not the model (number of conceptual categories). Therefore, we cannot expect future scaling of the workload to mitigate the poor parallel scalability of the SSI application.

B. Parallel Scalability of GLVQ

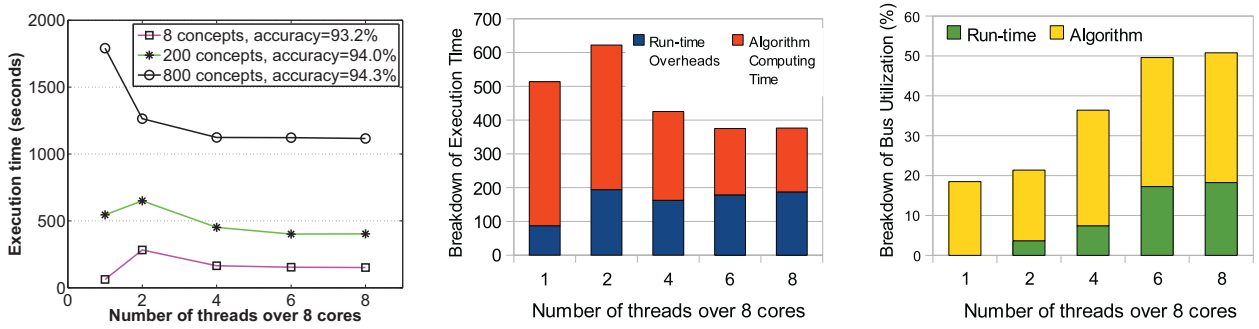
We first analyze the computational structure of the GLVQ training algorithm to identify opportunities for conventional parallelization that can be exploited using frameworks such as OpenMP [4], Intel's TBB [21], *etc.*

Figure 3 indicates that training vectors have to be processed sequentially since there are read-after-write (RAW) dependencies: the two reference vectors updated by the previous training vector are used in the distance calculation of the next training vector. There may also exist write-after-write (WAW) dependencies should the next training vector update the same reference vector(s) as the previous training vector. Nevertheless, the major component of the algorithm — the distance calculation between a training vector and all reference vectors — can still be parallelized across reference vectors or even across dimensions. In the case of eye detection using GLVQ, distance calculation can be parallelized across the 128 reference vectors each with a dimensionality of 512. This parallelism may seem sufficient to achieve speedup on an 8-core platform, however, as we show below that is not the case.

We implemented the GLVQ-based eye detection application described above in C++ and parallelized it based on the distance calculation between each training vector and the reference vectors. The parallel implementation was developed using Intel's Threading Building Blocks (TBB) framework [21]. We considered a data set of 2400 images of eyes and 3000 images of non-eyes for training. Images were selected from the Yale face database [23] in grayscale with a cropped size of 48×48. The parallel eye detection application was evaluated on a Dell Poweredge 2950 8-core machine (2-way SMP system with Intel Xeon E5320 quad-core CPUs) with 12 GB memory running RedHat Enterprise Linux 5.

[2]We use the same code for the single-threaded and multi-threaded versions, so even the 1 thread case incurs a small overhead due to initializing TBB.

[3]For the given multi-core platform, 60% utilization is the empirical limit beyond which memory latencies increase drastically [24].



(a) Parallel scalability (b) Run-time overheads with 200 concepts (c) Bus utilization with 200 concepts

Figure 4. Performance analysis of a conventional parallel SSI implementation

Figure 5(a) presents the performance of the parallel implementation as the number of threads varies from 1 to 8 (we stop at 8 threads since the underlying hardware platform is an 8-core system and further increase does not improve performance). We present results for two scenarios - when each class has 16 reference vectors (total of 32 reference vectors), and when each class has 64 reference vectors (total of 128). In the case of 64 reference vectors per class, we observe that the parallel implementation improves performance by only 1.6X from one to eight threads. Figure 5(a) also suggests that the fewer the reference vectors there are, the less scalable the performance is. Although this might lead us to suggest that increasing the number of reference vectors (beyond 64 per class) could lead to better scalability, in reality this only increases the overall workload and does not gain more accuracy due to the phenomenon of model over-fitting [11].

Further analysis of the parallel implementation using VTune [32] suggests that bandwidth is not the scaling bottleneck of GLVQ — as Figure 5(c) shows, the bus utilization is below 20% even with 8 threads. We further breakdown GLVQ's overall execution time in Figure 5(b). We observe that the proportion of time spent in run-time overheads increases from 4% to 54% as the application is scaled from 1 thread to 8 threads. The data suggest that the poor scalability is because the granularity of parallelism is too small to be profitably exploited on the given platform — each parallel task generated by the application represents only a relatively small amount of computation, and the run-time library and thread management overheads are significant compared to the task execution times.

C. The Scaling Bottlenecks

Analysis has shown that not all RM applications demonstrate parallel scalability over a larger number of cores. By partitioning the limited workload within every training instance, each parallel thread is frequently assigned a small amount of computation. Such a fine-grained parallel implementation leads to significant run-time overheads in the threading library. Moreover, the increase in the number of concurrent threads also results in more data requests within the same period of time. Consequently, the bandwidth demand may increase drastically and eventually limit scalability. We identified several properties that relates to the forgiving nature of RM workloads which entitle us to address these bottlenecks.

IV. Domain Characteristics

The previous section shows that RM workloads such as SSI and GLVQ demonstrate poor scalability on multi-core

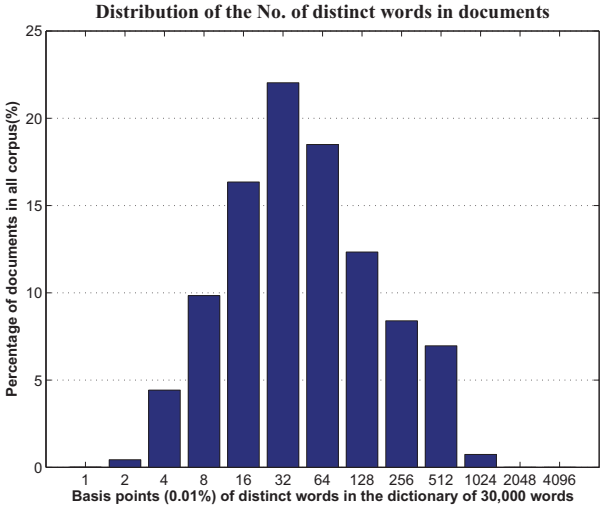
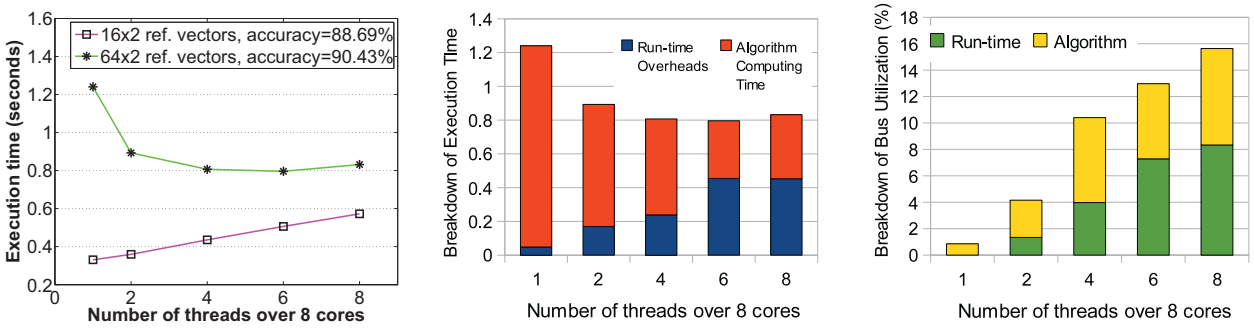


Figure 6. Distribution of number of distinct words in the documents used for SSI training, reflecting the fraction of the model updated in each training iteration. This does not illustrate the likeliness that subsequent iterations update the same part of the model, which will be shown in Figure 7(a).

platforms through conventional parallelization techniques. In this section, we describe the characteristics of these algorithms that form the basis for the proposed approach.

Both the algorithms described in Section III follow a computational template wherein a model or intermediate result is iteratively refined based on provided input training or testing data. The reads and writes to the model introduce data dependencies between successive "iterations". In conventional parallel implementations, iterations are therefore executed serially, and only parallelism within each iteration is exploited.

To further exploit parallelism beyond conventional techniques, we leverage the key domain characteristic of RM applications — their inherent forgiving nature. Most RM applications accept input data that is noisy and redundant, they perform computations that are statistical in nature, and they can produce a large number of numerically distinct solutions that are all considered equivalent or acceptable. These properties imply that *numerical equivalence between the parallel and sequential implementations is not mandatory*; different implementations are equivalent as long as they produce similar results of similar quality (classification accuracy, similarity scores, *etc.*). This enables us to improve performance beyond what can be achieved by conventional



(a) Parallel scalability

(b) Run-time overheads with 64×2 reference vectors

(c) Bus utilization with 64×2 reference vectors

Figure 5. Performance analysis of a conventional parallel implementation of GLVQ.

parallelization techniques.

We hypothesize that *judiciously relaxing or ignoring some of the data dependencies between iterations, thereby exploiting parallelism across iterations, will lead to significant improvements in parallel performance, without adversely impacting the output quality.*

To justify this hypothesis, we empirically analyze SSI and GLVQ applications to demonstrate three characteristics: write sparsity for an individual training iteration, low spatial locality of writes between training iterations, and the inherent error tolerance of the application.

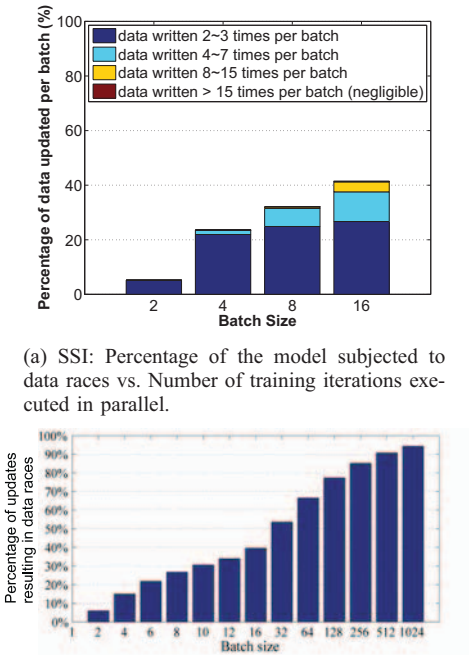
A. Write Sparsity for Individual Iterations

Both SSI and GLVQ algorithms demonstrate *write sparsity* — each training iteration only updates a small portion of the model. For SSI, some training iterations will not update the model at all — for all epochs after the 10^{th} epoch, only 28% of the training instances actually trigger backward propagation. Even for those training instances, the number of updated rows in U depends on the total number of distinct words with nonzero TFIDFs in the training documents (q, d, r). Figure 6 shows the distribution of number of distinct words that are present in the documents in the Wikipedia database. The figure shows that most documents use only a tiny part of the entire vocabulary. The number of distinct words in a document directly corresponds to the portion of the model (U) that may be updated when the document is used as training data in an iteration. On average, a training instance composed of three documents updates only 3.6% of the rows in U during backward propagation. In the case of eye detection using GLVQ, only two out of 128 reference vectors (or about 1.6% of the model) are updated in each iteration.

This write sparsity suggests that only a small part of data modifications in the next iteration depends on the values written in the previous iteration. It also suggests a possibility that consecutive iterations are not likely to update the same part of the model, which we then characterize quantitatively as the spatial locality of writes across iterations.

B. Low Spatial Locality of Writes Across Iterations

Write sparsity suggests that each iteration updates only a small part of the model. However, sparsity will not in itself be useful if consecutive iterations update exactly the same parts. Therefore, we need to study the locality of model updates across consecutive iterations. From a different perspective, when data dependencies are relaxed and training iterations are parallelized, the application may be subject to *data races* that would undermine output quality.



(a) SSI: Percentage of the model subjected to data races vs. Number of training iterations executed in parallel.

(b) GLVQ: Percentage of updates due to data races vs. Number of training iterations executed in parallel.

Figure 7. Characterizing spatial locality of writes across iterations.

In SSI, we estimate the number of potential data races by measuring the number of rows in U that are updated simultaneously by multiple threads [4]. According to the number of parallel threads, training iterations are batched and each batch is distributed across threads in parallel. The percentage of model updates subjected to data races is calculated by dividing the number of data races by the total number of updated rows in U for a given batch. Data is estimated using a trace of row numbers updated in each training iteration, generated using sequential code. When we hypothetically scale the number of parallel iterations to 8, only 32% of the rows updated by a batch may be subjected to data races. Figure 7(a) breaks down the percentage of rows updated by an average batch according the degree of data races they incur. We observe that even with eight threads,

[4]This is a worst-case value, since some of these potential data races could result in no numerical difference to the model if the timing of updates results in the same order as the sequential case.

very few rows (<7% of updated rows in U, or <1.3% of U) are updated by more than four threads.

For eye detection with GLVQ, we characterize the number of data races that could potentially occur if a batch of training instances is executed in parallel. Data is collected from a sequential execution trace, which is then used to measure the number of times that a reference vector is updated more than once within a given batch size. This number is divided by the total number of reference vector updates to compute a bound on the fraction of model updates that can be incorrect due to data races. Figure 7(b) illustrates the trend of percentage of model updates due to data races with the number of iterations executed in parallel, *i.e.*, window size. Clearly, larger windows expose greater parallelism across iterations, but also lead to more data races and potential for degradation in the quality of the result. The fraction of updates impacted by data races stays below 30% even with a window of 10 iterations. While this seems like a large number, as we show in the next sub-section, the errors due to these data races are easily absorbed by the inherently error resilient nature of the application, *e.g.*, the errors may be rectified by subsequent training iterations.

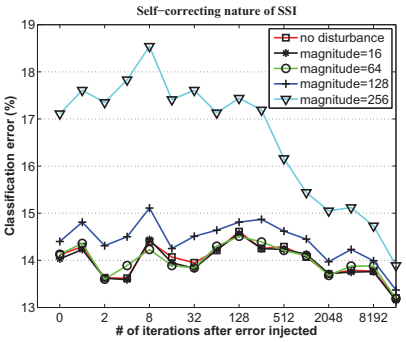
We note that the locality of updates is dependent on the order in which training vectors are processed. For the experiments described above, we did not alter the order in which training data was presented to the algorithms by the serial implementation of the applications. However, in Section VII, we describe how this can be leveraged to further mitigate the impact of data races due to dependency relaxation.

C. Tolerance to errors - the forgiving nature of RM algorithms

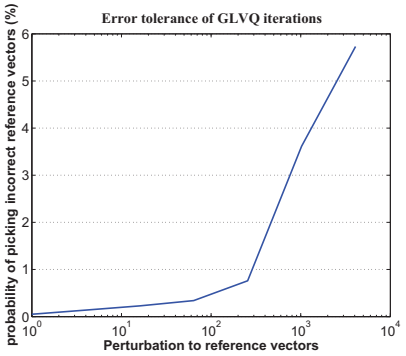
As described earlier in the paper, RM applications have an inherently *forgiving nature* due to a variety of reasons including redundancy and noisiness of the input data and statistical nature of the constituent computations. We attempt to characterize this forgiving nature by examining (i) how errors in the model due to data races impact a single iteration, and (ii) how subsequent iterations correct any residual errors resulting in a self-healing behavior.

We characterize the self-correcting behavior of the SSI algorithm by disturbing the updated rows of U in the 10000^{th} training iteration, and measuring the application-level error rate (accuracy of the model in identifying similar documents) after allowing varying number of training iterations to "correct" the error. The results are presented in Figure 8(a). We observe that the effects of model perturbation are hardly noticeable until the error magnitude reaches 128 times the average change in model updates. Moreover, even errors of such large magnitude can be eventually corrected by later iterations.

In the case of eye detection using GLVQ, we observed that the reference vectors are updated with an average offset of 0.135 and a standard deviation of 0.006 (for a sequential implementation). We perturb the reference vectors during training by a range of magnitudes at the beginning of an iteration and computed the probability that the iteration would pick an incorrect reference vector to update as a result of the perturbation. Figure 8(b) shows the results of this experiment, and indicates that an iteration will rarely pick an incorrect reference vector to update, unless the disturbance to the current reference vector is more than 128 times larger than normal changes in model updates. We note that in practice, the "errors" introduced in a model due to data races are unlikely to be so large due to the "converging" nature of the model values. As a result, the occasional inaccuracy introduced to a portion of the model due to data races is not likely to propagate to other parts of the model.



(a) SSI: Application-level error rate due to perturbation of the model vs no. of iterations after the perturbation.



(b) GLVQ: Probability of picking an incorrect reference vector in an iteration for different magnitudes of perturbation.

Figure 8. Characterizing error tolerance.

D. Comparing sequential and parallel implementations

The characteristics identified in this section suggest that for the considered application domain, we could explore parallel implementations that do not preserve numerical equivalence with the sequential implementation. We argue that this is perfectly acceptable for RM applications, where there is often no "golden" result. For example, providing input data in a different order leads to a different result even for the sequential implementation. This provides a natural tolerance range within which all outputs could be considered equivalent.

Our objective is to show that proposed method improves performance for comparable quality. Our paper compares sequential and parallel implementations in two different ways:

- Quality of solutions must be *identical*: performance comparison between different methods are not based upon time spent for a fixed amount of work, but time spent to achieve a fixed accuracy (Section VI). This is possible since many applications (including the two considered in this paper) iteratively refine the model until a specified accuracy is achieved. Under this comparison scenario, the proposed techniques may slightly increase the number of iterations that the algorithm needs to execute; however, this increase is significantly outweighed by the greater efficiency in executing each iteration.
- Quality of solutions is *comparable*: performance comparison between different methods are based upon time

spent for a fixed amount of work, with the precondition that error in the output falls within a small tolerance range (Section V and VII). The tolerable range of error is obtained empirically from the sequential implementation.

V. Data Dependency Relaxation

The discussions in Section IV demonstrate that data dependencies among iterations are sparse and can be relaxed without significantly impacting the accuracy of the resulting model. Therefore, we propose an alternative approach to parallelizing RM algorithms that creates and exploits parallelism across iterations through data dependency relaxation. By allowing each thread to process different training instances independently rather than process only a fraction of a training instance, we create parallelism at a coarser granularity with minimal communication and synchronization overheads.

Although we demonstrate this approach on two examples, we believe that our technique is applicable to a broader class of applications, and it is most effective under the following circumstances:

- When the training instances are small or sparse (e.g. SSI), or when the size of the model is small (*e.g.*, eye detection using GLVQ), parallelizing the processing of a single training instance cannot be profitably exploited on current multi-core platforms using conventional parallel implementations.
- When the number of training instances is large, relaxing the dependency among them may increase the granularity of parallelism significantly.
- Each training instance updates only a small portion of the model therefore error is less likely to accumulate and can be self-corrected by later iterations.

Therefore, we believe that the same principles have potential for application to other RM algorithms such as multilayer neural networks [29], Support Vector Machines [10], and Hidden Markov Models [14].

Figure 9 presents data dependency relaxed versions of the SSI and GLVQ algorithms using the iterative-convergence programming template designed for RM workloads [25]. The iterative-convergence template naturally captures algorithms that perform a parallel computation repeatedly until a specified convergence criterion is reached. The parallel computation may be specified using constructs such as `parallel_iterate` and `parallel_reduce`. In our implementation of the SSI algorithm, the `parallel_iterate` represents one *epoch* of training corresponding to 10,000 iterations, and is followed by a `parallel_reduce` that computes the error on a randomly selected testing set. The `parallel_reduce` computation is straightforward to parallelize efficiently. Our focus is on the `parallel_iterate`, which is difficult to parallelize due to the dependencies between iterations induced by the model updates. In the case of the GLVQ algorithm for eye detection, the `parallel_iterate` represents the entire training algorithm, and the convergence condition is specified to TRUE indicating that only one pass is made through the training data.

For data dependency relaxation, the `batch` keyword is used to specify that multiple iterations may be executed in parallel while relaxing data dependencies between them. The parameter to this keyword, the batch size, specifies the maximum number of training instances that may be subject to dependency relaxation.

A. Improvements in Performance Scalability

To justify our hypothesis that relaxing data dependencies among training iterations improves performance scalability while still generating acceptable accuracy, we implemented dependency relaxed versions of both SSI and GLVQ algorithms. We refer to this implementation as *DR* and the conventional parallel implementation as *Conventional*. By ignoring data dependencies, multiple parallel threads may update the same part of the model simultaneously, resulting in data races and it is not deterministic which value the model will end up with. In all our experiments, each run is repeated 5 times for SSI [5] and 100 times for GLVQ to even out measurement errors introduced by run-time variability.

We consider the output model as acceptable if it falls within a small, empirical error-tolerance range (less than 0.29%). The tolerance range is empirically obtained as the variation in accuracy when the conventional implementation is subjected to different orders in which input training instances are considered. Note that the inherent nature of RM algorithms is different from typical numerical algorithms; even for the conventional implementation, there is no "golden" result.

As a result of data dependency relaxation, SSI becomes more scalable. Figure 10(a) compares the result of *Conventional* and *DR* implementations. The speedup resulting from the *DR* case with 8 threads is 3.1X instead of 1.3X with the *Conventional* case. On the other hand, modeling accuracy drops very minimally, from 93.95% to 93.84%. Further analysis shows that while the run-time overhead is drastically reduced and is no longer the bottleneck (Figure 10(b)), bus utilization increases close to 60% at six threads, as shown in Figure 10(c). According to the VTune Performance Analyzer, a bus utilization larger than 60% leads to performance degradation, and it keeps SSI from further performance scalability.

After data dependency relaxation, the eye detection application using GLVQ shows significant improvement in performance and parallel scalability. As Figure 11(a) shows, the speedup resulting from 8 threads is 7.3X instead of 1.49X for the *Conventional* case. The accuracy only decreases from 90.44% to 90.41%, which is negligible. As Figure 11(b) demonstrates, the *DR* implementation overcomes the bottleneck of run-time overheads that is caused by the small amount of workload in each task in the conventional parallel implementation. In the meantime, although the memory bandwidth demand of GLVQ increases with more parallel threads, the bus utilization for the 8-thread case is still well below 60% (Figure 11(c)) and is far away from impacting application scalability. As a result, the performance scaling of the *DR* implementation is close to linear.

VI. Computation Dropping for Memory- and Coherence- Traffic Reduction

Data-dependency relaxation enables SSI to exploit more parallelism and reduce run-time overhead. However, with 8 threads, the bus utilization approaches 60% in both *Conventional* and *DR*, which indicates that bandwidth may penalize performance and become the scaling bottleneck [24]. VTune analysis reveals that *DR* scales better than *Conventional* also because it dedicates almost all of the bus utilization to the SSI algorithm, while in *Conventional* the run-time system consumes a significant amount of bandwidth and the SSI algorithm only utilizes 32.5% of the bus (Figure 4(c)). Yet, although *DR* allows the bus to be solely utilized by the SSI algorithm, it cannot go beyond this bandwidth limitation.

In our SMP system with two quad-core sockets, bus utilization can be attributed to two factors: memory traffic and coherence traffic. In fact, since the quad-core processors

[5] We performed fewer repetitions for SSI due to the significantly longer execution time.

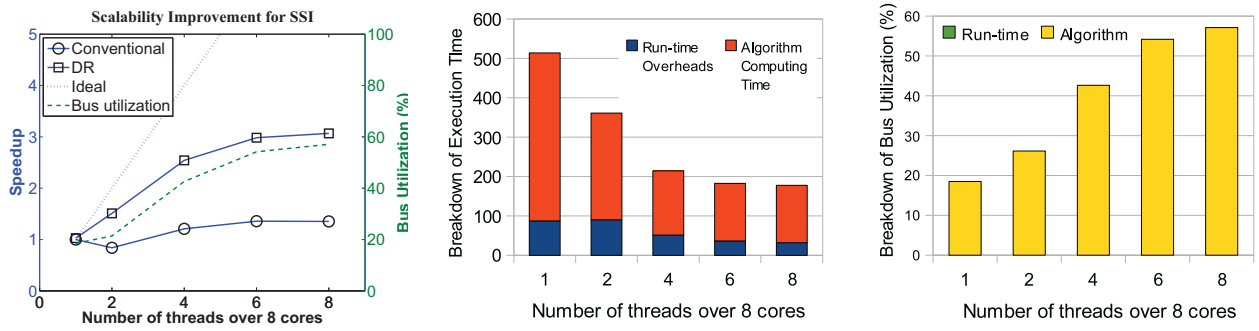
```
int N, C;                /* N words in vocabularies, C concepts */
double U[N][C];          /* weight matrix                       */
int count=0;             /* number of trained instances         */
int E;                   /* epoch length                        */
int threshold;           /* threshold of error                  */
SSI_train()
{
 iterate
 {
    /* train 10K instances */
    parallel_iterate(i = count to count+E batch P)
    {
       /* q: query document                */
       /* d: positively related document   */
       /* r:  irrelevant document          */
       q, d, r = random_tuple();
       U, bias = forwardBackward(q, d, r, U, bias);
    }
    /* test 10K instances */
    parallel_reduce(i = count to count+E)
    {
       q, d, r = random_tuple();
       error += forward(q, d, r);
    }
    count += E;
 } until converged (error < threshold);
}
```

(a) Expressing Data-dependency Relaxation in SSI

```
int C, N;                /* C classes, N training vectors */
int M;                   /* M reference vectors per class */
ref_vecs [1:C][1:M];     /* reference vectors */
train_vecs [1:N];        /* training vectors */
labels[1:N];             /* class labels for each training vector */
int P;                   /* No. of training vectors to process in parallel */

GLVQ_train_best_effort()
{
 iterate
 {
    parallel_iterate(i = 1 to N batch P)
    {
       /* calculate distances from a training vector to all reference vectors */
       distances[1:C][1:M] = euclid(ref_vecs[1:C][1:M],train_vecs[i]);

       /* pick two reference vectors */
       /* one from correct class, one from other classes */
       target_ref, other_ref = choose_nearest_vectors(labels[i],
          ref_vecs[1:C][1:M], distances[1:C][1:M]);

       /* update the chosen reference vectors */
       update_refs(target_ref, other_ref, distances[1:C][1:M]);
    }
 } until converged (TRUE); /* only one pass through training vectors */
}
```

(b) Expressing Data-dependency Relaxation in GLVQ

Figure 9. Data-dependency relaxation can be easily expressed in both SSI and GLVQ using the best-effort programming template.

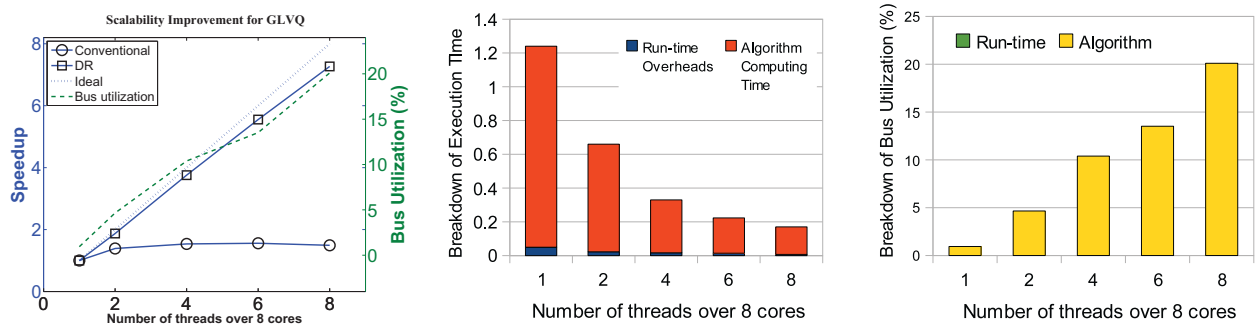


(a) Speedup vs No. of threads. The overall utilization of DR is also shown

(b) Performance breakdown of DR

(c) Bus utilization of DR

Figure 10. Performance analysis of a dependency relaxed parallel implementation of SSI. The training process takes 100 epochs each has 10,000 training instances.



(a) Speedup vs No. of threads. The overall utilization of DR is also shown

(b) Performance breakdown of DR

(c) Bus utilization of DR

Figure 11. Performance analysis of a dependency relaxed parallel implementation of GLVQ.

used (Intel Xeon E5320) are composed of two dual-core dies integrated into the same package, some of the coherence traffic between two cores on the same chip also goes through the off-chip system bus. We investigate how each factor (memory traffic and coherence traffic) affects SSI's parallel scalability by performing hypothetical sensitivity studies regardless of the output accuracy. To remove the majority of memory traffic, we use only a subset of training data that would fit in the last level cache. This increases the parallel scalability from 3.1X to 3.5X when 8 threads are used. To further remove coherence traffic which is mainly caused by modifying shared data, some local temporary variables are modified instead of the shared model. This results in a parallel scalability of 6.7X with 8 threads, which

indicates that while the bandwidth bottleneck is caused by a combination of memory- and coherence-traffic, the latter may be a more significant contributor.

To overcome this bandwidth bottleneck, the forgiving nature of RM algorithms can be further exploited by selectively dropping noncritical memory operations that consume bus bandwidth. As a result, the limited bandwidth can be dedicated to process data with more significance — in the case of SSI, this would be words with larger TFIDFs. Characterizations show that a significant number of words have low TFIDFs between 0 and 0.01 while around 10% TFIDFs can reach 0.06 or higher, as shown in Figure 12(a).

We propose *DR-DropWrd* that drops memory operations and the corresponding computation associated with TFIDFs below a threshold in both forward and backward propagation. In other words, given words with low TFIDFs in a training document, their corresponding rows in the model U are not fetched or updated, saving both memory- and coherence-traffic.

To further reduce shared data modifications and save coherence traffic during backward propagation, we attempt to update an even smaller part of the model which corresponds to words with significantly large TFIDFs. This technique is built upon *DR-DropWrd* and is named as *DR-DropCoh*. Compared to *DR*, both *DR-DropWrd* and *DR-DropCoh* produce models with less accuracy given the same number of training instances, because fewer words are used in each document. To achieve the same accuracy of 95%, the required number of epochs is 121 on average for conventional implementation, 148 for *DR-DropWrd*, and 167 for *DR-DropCoh*. However, by iterating through more training instances and processing each instance faster, both *DR-DropWrd*, and *DR-DropCoh* can achieve the same accuracy as *DR* in less amount of time.

We fix the accuracy that must be achieved by all methods, and then compare the performance of various methods. Figure 12(b) compares *DR*, *DR-DropWrd* and *DR-DropCoh* with regarding to their execution time required to train a model with 95% accuracy. In this experiment, *DR-DropWrd* skips both forward and backward propagation associated with TFIDFs lower than 0.01; *DR-DropCoh* skips words with TFIDFs lower than 0.01 during the forward propagation and only update the rows in U that corresponding to words with TFIDFs larger than 0.06. We show that both *DR-DropWrd* and *DR-DropCoh* achieve the same accuracy within less time than *DR* — both leads to around 36% performance gains during sequential execution. *DR-DropCoh* is more effective in reducing coherence traffic which in turn benefits parallel execution, and it leads to larger performance gains of 59% in the case of 8 threads compared to *DR*. Using such a combination of data-dependency relaxation and traffic reduction techniques, a speedup of 5.5X is achieved compared to the conventional sequential execution. Further analysis using VTune shows that with 8 threads, *DR-DropCoh* is able to produce an identically accurate model with 39% less bus transactions (Figure 12(c)) than *DR*. While *DR-DropWrd* and *DR-DropCoh* are SSI-specific optimizations, the same principle applies to other bandwidth bounded applications with similar forgiving nature.

VII. Error Mitigation

As discussed in Section V, relaxing data dependencies incurs data races. This could impact the training algorithm in the following ways:

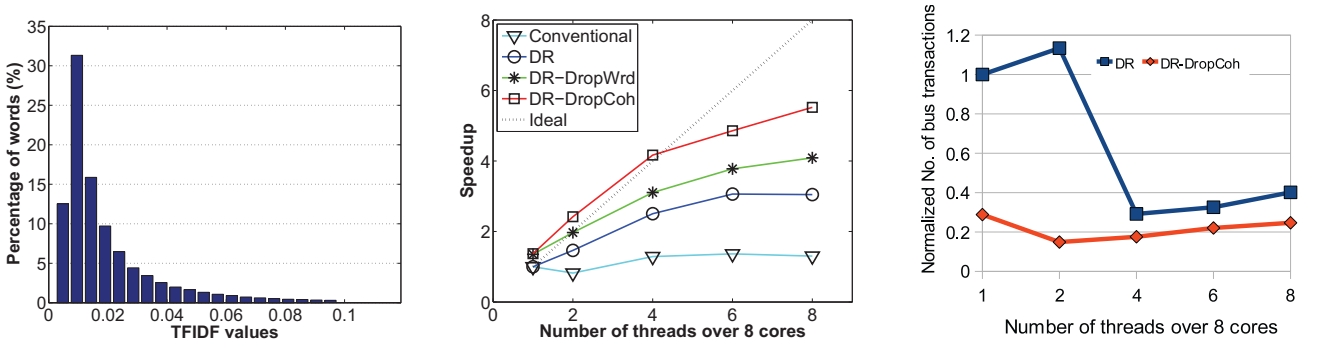
- *Partially written data structures.* When multiple threads update the same reference vector in GLVQ or the same row of U in SSI, it is possible that these data structures end up with some values computed from one thread and other values computed elsewhere.
- *Partially read data structures.* For GLVQ, when one thread writes to a reference vector while another reads from it, the latter thread may read some obsolete values and some updated values. The same thing may happen to rows of U in the case of SSI.
- *Conflicted learning.* The rationale behind refining the model in a sequential way is that each training instance performs an expectation-maximization stage based on the previously trained model. However, when several training instances are processed in parallel, they may attempt to refine the same previous model in different, even contradictory, ways. Accumulating their updates without precaution may cancel out their effects.

Embracing and exploiting the forgiving nature of RM applications is the key basis for our work, therefore fully eliminating the numerical errors that are caused by dependency relaxation would be both unnecessary and excessive. Instead, we introduce several strategies to partially mitigate the aforementioned effects, with an objective to provide different tradeoffs between performance and output quality. These techniques lie in between the two extremes of conventional parallel implementations and fully dependency relaxed implementations.

- *Atomic write.* Mutex locks are used to ensure that all values within an individual data structure (*e.g.*, a reference vector in GLVQ or a row of U in SSI) always conform to the updates by the same training instance. If multiple threads attempt to update the same data structure simultaneously, only one of them proceeds and others have to wait.
- *Atomic read.* Mutex locks are used to ensure that individual data structures are not partially written while they are being read or vice versa.
- *Conflict detection and recovery.* Parallel training iterations are instrumented with the capability to detect when they attempt to update the same part(s) of the model. When this happens, only one of them proceeds; others are aborted and re-scheduled for execution at a later time. Since the conflicting update is detected before the model is written, rollback is unnecessary for aborted instances. Note that this reduces but does not eliminate conflicted learning. Conflicts are not detected if two parallel instances write to the same part of the model at separate times — in such cases, the updates of one training instance will overwrite the updates from the other instance.

In Figures 13 and 14, *DR-Atomic* refers to the implementation where individual data structures are guaranteed to be read and written atomically; conflicting accesses are queued and resolved later. *DR-Reschedule* refers to the implementation where conflicting updates are detected and entire training iterations are aborted and restarted later. In GLVQ, parallel training instances seldom update the same reference vectors simultaneously, therefore conflicts rarely take place and this justifies the use of *DR-Reschedule*. On the other hand, characterization of SSI in Section IV-B shows that two training instances are likely to update some common rows in U, although the conflicted rows account only for a small portion of updates. In such cases, the *DR-Reschedule* implementation does not improve performance since it tends to serialize all training instances similar to the *Conventional* case (therefore, it is not shown in Figure 13).

In the case of GLVQ with 64 reference vectors for each class, the *DR-Atomic*, *DR-Reschedule*, and *DR* implementations achieve almost the same accuracy since the accuracy of DR is already close to the *Conventional* case. Figure 14

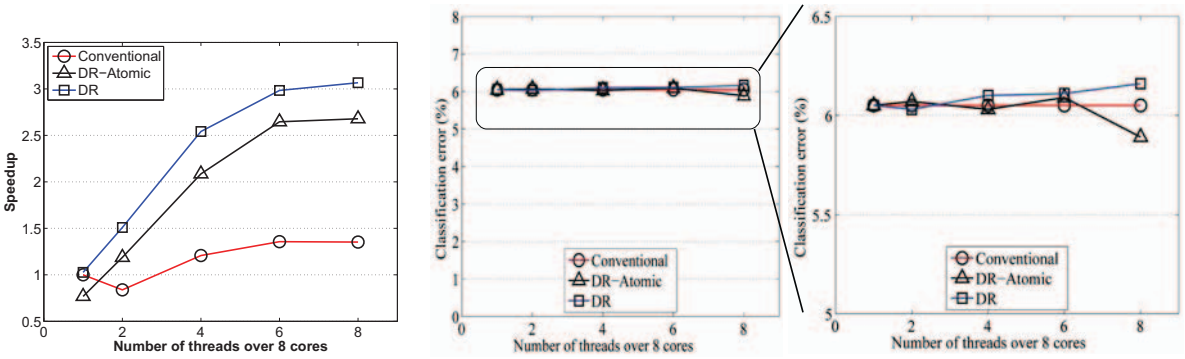


(a) Distribution of TFIDFs

(b) Performance scaling when data-dependency relaxation is combined with the traffic reduction technique.

(c) Reduction in total number of bus transactions

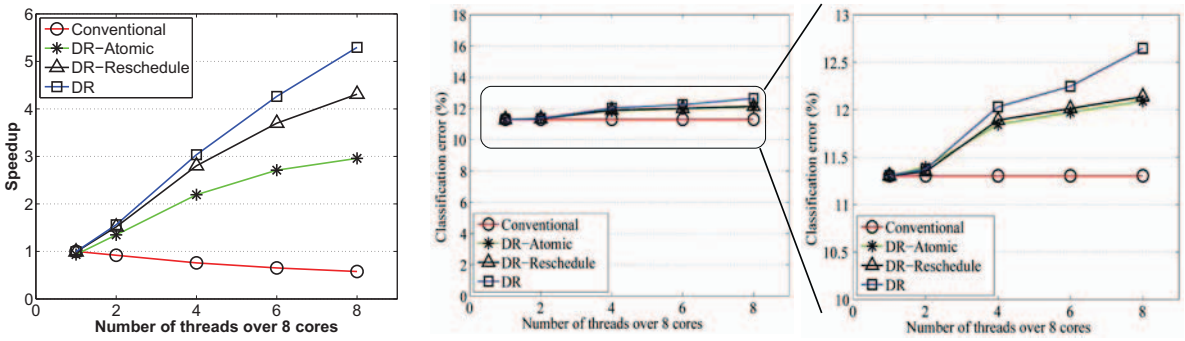
Figure 12. Performance analysis of traffic reduction.



(a) Performance scaling

(b) Accuracy tradeoff

Figure 13. Performance scaling and accuracy tradeoff in SSI resulting from relaxing data dependency



(a) Performance scaling

(b) Accuracy tradeoff

Figure 14. Performance scaling and accuracy tradeoff in GLVQ resulting from relaxing data dependency. Each class has 16 reference vectors.

shows the results when we reduce the number of reference vectors per class from 64 to 16 in GLVQ. While *DR-Atomic* and *DR-Reschedule* exhibit similar performance scalability to the *DR* case, they reduce the error rate from 12.7% to 12.1%. Figure 13 demonstrates the same phenomenon for SSI; once again, *DR-Atomic* and *DR-Reschedule* tradeoff performance scalability for more modeling accuracy. We expect that the proposed error mitigation techniques will be useful in guarding against potential accuracy losses due to more aggressive data dependency relaxation, when scaling to platforms with larger numbers of cores (*e.g.*, GPUs).

VIII. Related Work

RM workloads that have abundant parallelism have been parallelized in conventional ways using different programming models, including OpenMP [22] and MapReduce [9]. These workloads can also benefit from massively parallel platforms such as Graphics Processors (GPUs) [5], [7]. Application-specific algorithmic optimizations have also been exploited for better parallel implementation [18], [13]. However, these techniques do not address the poor performance scalability of those RM workloads that learn a small model or exhibit sparsity in training.

Speculative multi-threading [3], [27] extracts parallelism by out-of-order execution of the original thread. It relaxes control dependencies only and it may increase the overall

workload if speculation fails. Data speculation [19] and value prediction [16] do not break data dependencies as well — the result is numerically unchanged and the order of writes has to be preserved to be the same as the sequential code. By exploiting the forgiving nature of RM workloads, we are able to break this bottleneck to gain further scalability.

Relaxing data-dependency has only been used previously in iterative stencil loops [6], [15], [31] where data exchanges across iterations are sometimes skipped to reduce synchronization overhead, and obsolete data copies may be used instead. Our technique addresses a different problem where we reduce the run-time threading overhead by using data-dependency relaxation to restructure the parallel computation and partition the workload on a coarser granularity in spite of occasional data races. This also differentiates our work from transactional memory where accuracy is guaranteed by rolling back upon data races [20]. Our software-based implementation is also much simpler than that of transactional memory.

IX. Conclusions

Recognition and Mining are expected to be ubiquitously used in a wide range of future computing applications, making it critical to ensure that they can scale in performance on multi-core and many-core computing platforms. While some RM workloads do scale very well with increasing number of cores, others do not, due to a variety of factors including limited granularity of parallelism, and overheads of synchronization/communication/off-chip bandwidth. In this paper, we presented a domain-specific approach to parallelize a class of Recognition and Mining algorithms. Our approach leverages the unique characteristics of RM applications, and hence goes beyond conventional parallelization techniques that focus on numerical equivalence of the parallel implementation and the sequential implementation. The proposed technique creates parallelism at a coarse granularity by judiciously relaxing data dependencies across iterations to reduce run-time overheads. We demonstrated that this approach does not significantly alter the quality of the final output, and presented techniques to mitigate any degradation in output quality. Finally, for RM workloads whose scalability is limited by the bandwidth bottleneck, their performance and scalability can be further improved by selectively dropping non-critical memory operations that contribute significantly to memory- and coherence- traffic.

Acknowledgments: We would like to thank several members of the Machine Learning Department at NEC Labs America, notably Bing Bai, Hiroyoshi Miyano, Jason Weston, and Ronan Collobert, and Hans Peter Graf for providing us the applications used in this work.

References

[1] Wikipedia page for TF-IDF. http://en.wikipedia.org/wiki/Tf-idf.

[2] Bing Bai, Jason Weston, Ronan Collobert, and David Grangier. Supervised semantic indexing. In *ECIR*, pages 761–765, 2009.

[3] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *SPAA*, pages 99–108, 2002.

[4] OpenMP Architecture Review Board. OpenMP application program interface. http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[5] B. C. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. *ICML*, 2008.

[6] D. Chazan and W. Miranker. Chaotic relaxation. Linear algebra and its applications. 2(2):199–222, 1969.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. *JPDC*, 2008.

[8] Y. K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, M. Smelyanskiy, and M. Smelyanskiy. Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications. *Proc. of IEEE*, 96:790–807, 2008.

[9] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS*, pages 281–288, 2006.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[11] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[12] Pradeep Dubey. A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *White Paper, Intel Corporation*, 96, 2008.

[13] I. Durdanovic, E. Cosatto, and H.P. Graf. Large-scale parallel implementations of SVMs. *Large Scale Kernel Machines*, 2007.

[14] Shai Fine and Yoram Singer. The hierarchical hidden markov model: Analysis and applications. In *MACHINE LEARNING*, pages 41–62, 1998.

[15] A. Frommer and D. B. Szyld. On asynchronous iterations. *J. Comp. Appl. Math.*, 213(1–2):201–216, Nov 2000.

[16] Freddy Gabbay and Avi Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Trans. Comput. Syst.*, 16(3):234–270, 1998.

[17] R. Gleim, A. Mehler, and M. Dehmer. Web Corpus Mining by instance of Wikipedia. *Web as Corpus*, 2007.

[18] Hans Peter Graf, Eric Cosatto, Leon Bottou, Igor Durdanovic, and Vladimir Vapnik. Parallel Support Vector Machines: The cascade SVM. In *NIPS*, pages 521–528, 2005.

[19] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, 1998.

[20] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *ISCA*, 21(2):289–300, 1993.

[21] Intel Corporation. Intel Threading Building Blocks. http://www.threadingbuildingblocks.org.

[22] A. Jaleel, M. Mattina, and B. Jacob. Last-level cache (LLC) performance of data-mining workloads on a CMP–A case study of parallel bioinformatics workloads. In *HPCA*, 2 2006.

[23] David Kriegman and Peter Belhumeur. The Yale face database. http://cvc.yale.edu/projects/yalefaces/yalefaces.html.

[24] R. K. Malladi. Using Intel Vtune^{TM} Performance Analyzer Events/Rations and Optimizing Applications. *White Paper, Intel Corporation*, 2009.

[25] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. Best-effort parallel execution framework for recognition and mining applications. *IPDPS*, 2009.

[26] Nikhil R. Pal, James C. Bezdek, and Eric C.-K. Tsao. Generalized clustering networks and Kohonen's self-organizing scheme. *Neural Networks, IEEE Transactions on*, 4(4):549–557, Jul 1993.

[27] Carlos García Qui nones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI*, pages 269–279, 2005.

[28] Atsushi Sato, Hitoshi Imaoka, Tetsuaki Suzuki, and Toshinori Hosoi. Advances in face detection and recognition technologies. *NEC Journal of Advanced Technology, Winter 2005*.

[29] Murray Smith. *Neural Networks for Statistical Modeling*. John Wiley & Sons, Inc., 1993.

[30] Katsuhiko Takahashi and Daisuke Nishiwaki. A class-modular GLVQ ensemble with outlier learning for handwritten digit recognition. In *ICDAR*, page 268, 2003.

[31] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *ICS*, pages 244–255, 2009.

[32] Alexander Wolfe. Toolkit: Intel's heavy-duty dev tools. *Queue*, 2(2):12–17, 2004.