Parallelizing Word2Vec in Shared and Distributed Memory

Shihao Ji, Nadathur Satish, Sheng Li, Pradeep Dubey
Parallel Computing Lab, Intel Labs, USA
Emails: {shihao.ji, nadathur.ragopalan.satish, sheng.r.li, pradeep.dubey}@intel.com

Abstract—Word2Vec is a widely used algorithm for extracting low-dimensional vector representations of words. It generated considerable excitement in the machine learning and natural language processing (NLP) communities recently due to its exceptional performance in many NLP applications such as named entity recognition, sentiment analysis, machine translation and question answering. State-of-the-art algorithms including those by Mikolov et al. have been parallelized for multi-core CPU architectures but are based on vector-vector operations that are memory-bandwidth intensive and do not efficiently use computational resources. In this work, we improve reuse of various data structures in the algorithm through the use of minibatching, hence allowing us to express the problem using matrix multiply operations. We also explore different techniques to parallelize word2vec computation across nodes in a compute cluster, and demonstrate good strong scalability up to 32 nodes.

I. INTRODUCTION

Natural language processing (NLP) aims to efficiently process text information and enable understanding of human language; this is one of the most critical tasks toward artificial intelligence [1]. One of the fundamental issues in NLP concerns how to represent words of a language in machine, upon which more complex learning and inference tasks can be built efficiently. Instead of a traditional bag of words (or one-hot) encoding, distributed word embeddings represent each word as a low-dimensional dense vector in such a way that semantically and syntactically similar words are close to each other. This idea has been applied to a wide range of NLP tasks with considerable success [2], [3], [4].

Recently, Mikolov et al. [5] generated considerable excitement in the machine learning and NLP communities by introducing a neural network based model to learn distributed word representations, which they call word2vec. It was shown that word2vec produces state-of-the-art performance on word similarity, word analogy tasks as well as many downstream NLP applications such as named entity recognition, machine translation and question answering [6], [7]. The word similarity task is to retrieve words that are similar to a given word. On the other hand, word analogy requires answering queries of the form $a:b:c:?$, where $a$, $b$, and $c$ are words from the vocabulary, and the answer to the query must be semantically related to $c$ in the same way as $b$ is related to $a$. This is best illustrated with a concrete example: Given the query $king:queen:man:?$ we expect the model to output $woman$.

The goal behind word2vec is to find word representations that are good at predicting nearby words from a training corpus. A common approach is to use the Skip-gram model with negative sampling [5]. This method involves judging similarity between two words as the dot product of their word representations, and the goal is to minimize the distance of each word with its surrounding words while maximizing the distances to randomly chosen set of words (a.k.a "negative samples") that are not expected to be close to the target.

The formulation of word2vec uses Stochastic Gradient Descent (SGD) to solve this optimization problem. SGD solves optimization problems iteratively; at each step, it picks a pair of words: an input word and another word either from its neighborhood or a random negative sample. It then computes a gradient of the objective function with respect to the two chosen words, and updates the word representations of the two words based on the gradient values. The algorithm then proceeds to the next iteration with a different word pair being chosen.

The formulation above has two main issues:

1) SGD is inherently sequential: since there is a dependence between the update from one iteration and the computation in the next iteration (they may happen to touch the same word representations), each iteration must potentially wait for the update from the previous iteration to complete. This does not allow us to use the parallel resources of the hardware.

2) Even if the above problem is solved, the computation performed in each iteration is a single dot product of two word vectors. This is a BLAS1 operation and is limited by memory bandwidth, thus not utilizing the increasing computational power of modern multi-core and many-core processors.

To solve (1), word2vec uses Hogwild [8], a scheme where different threads process different word pairs in parallel and ignore any conflicts that may arise in the model update phases. In cache-coherent architectures, however, Hogwild tends to have true and false sharing of the model data structure between threads, and is heavily limited by inter-thread communication.

In this work, we propose a simple yet efficient parallel algorithm to speed up the word2vec computation in shared memory and distributed memory systems.

- We present a scheme based on minibatching and shared negative sampling to convert the BLAS1 operations of
word2vec into BLAS3 matrix multiply operations, hence efficiently leveraging the vector units and multiply-add instructions of modern architectures. This is described in Section III.

- We parallelize this approach across batches of inputs, thereby reducing the total number of model updates to the shared model and hence limiting inter-thread communication. This allows our scheme to scale better than Hogwild.

- We perform experiments to scale out our technique to a cluster of 32 compute nodes. Nodes across the cluster perform synchronous model updates, and we follow the technique proposed in [9] to reduce network traffic. We overlap computation and communication to improve network utilization, and adjust the frequency of propagating model updates across the network to achieve balance between computation and communication. We adjust algorithmic learning rates to allow for good convergence in the presence of a limited number of updates.

In combination, these techniques allow us to scale the computation near linearly across cores and nodes, and process hundreds of millions of words per second, which is the fastest word2vec implementation to the best of our knowledge.

The remainder of the paper is organized as follows. In Sec. II we describe the basic word2vec model and the original parallelization scheme proposed by Mikolov et al. [5]. A new parallelization scheme is then presented in Sec. III, along with a distributed implementation cross nodes in a compute cluster. Example results on billions of words training corpora are presented in Sec. IV, with comparisons to the best known performance reported currently in the literature. Conclusions and future work are discussed in Sec. V.

II. THE WORD2VEC MODEL

Word2vec represents each word \( w \) in a vocabulary \( V \) as a low-dimensional dense vector \( v_w \) in space \( \mathbb{R}^D \), and attempts to learn the continuous word vectors from raw text such that the spatial distance between words then describes the similarity between words, e.g., the closer two words are in the embedding space, the more similar they are semantically and syntactically. These word representations are learned based on the distributional hypothesis [10], which assumes that words with similar context tend to have a similar meaning. Under this hypothesis, two distinct model architectures: Contextual Bag-Of-Words (CBOW) and Skip-Gram with Negative Sampling (SGNS) are proposed in word2vec to predict a target word from surrounding context [11], [5]. We focus here on the SGNS model since it produces state-of-the-art performance and is widely used in the NLP community.

The training objective of the Skip-gram is to find word representations that are useful for predicting the surrounding words in a sentence from a large textual corpus. Given a sequence of training words \( \{w_1, w_2, \cdots, w_T\} \), the objective of the Skip-gram model is to maximize the average log probability

\[
J(\Omega) = \frac{1}{T} \sum_{t=1}^{T} \sum_{c \subseteq \Omega \setminus c,j \neq 0} \log p(w_{t+j} | w_t),
\]

where \( \Omega \) is the model parameters to be trained (which will be defined soon), \( c \) is the size of the training context (a sliding window around the center word \( w_t \)), and \( p(w_{t+j} | w_t) \) is the probability of seeing word \( w_{t+j} \) given the center word \( w_t \), and is defined by a simplified one hidden layer neural network model, depicted in Fig. 1. The network has an input layer, a hidden layer without nonlinear transformation (also called projection layer), and a few softmax output layers corresponding to words within the context window. Typically, the network is fed as input \( w_t \in \mathbb{R}^V \), where \( V \) denotes the vocabulary size, and it produces a hidden state \( h \in \mathbb{R}^D \), where \( D \) is the size of the hidden layer or the dimension of the embedding space, which is in turn transformed to the output \( w_{t+j} \in \mathbb{R}^V \). Different layers are fully connected, with the weight matrix \( M_{out} \) at output layers shared among all context words. Collecting all the weight matrices from this architecture, we denote the model parameter by \( \Omega = \{ M_{in}^{V \times D}, M_{out}^{V \times D} \} \).

In the Skip-gram model above, the input \( w_t \) is a sparse vector of a 1-of-\( V \) (or one-hot) encoding with the element corresponding to the input word \( w_t \) being 1 and the rest of the components set to 0. Therefore, the basic Skip-gram formulation defines \( p(w_{t+j} | w_t) \) as the softmax function:

\[
p(w_t | w_t) = \frac{\exp(\langle v_{w_t}^{in}, v_{w_t}^{out} \rangle)}{\sum_{w=1}^{V} \exp(\langle v_{w}^{in}, v_{w}^{out} \rangle)}
\]

where \( \langle \cdot, \cdot \rangle \) denote the inner product between two vectors, \( v_{w_t}^{in} \) and \( v_{w_t}^{out} \) are the “input” and “output” vector representations of \( w \), corresponding to the respective rows of model parameter matrices \( M_{in} \) and \( M_{out} \). The computation of this formulation is prohibitively expensive since its cost is proportional to \( V^2 \), which is the size of the vocabulary and is often very large (e.g., around \( 10^6 \)).

To improve performance of word2vec, Mikolov et al. [5] introduced negative sampling that approximates the log of
softmax (2) as
\[
\log p(w_o|w_I) \approx \log \sigma(\langle v_{in}^w, v_{out}^w \rangle) + \sum_{k=1}^{K} \mathbb{E}_{w_k \sim P_n(w)}[\log \sigma(-\langle v_{in}^w, v_{out}^w \rangle)],
\]
where \(\sigma(x) = \frac{1}{1+\exp(-x)}\) is the sigmoid (logistic) function, and the expectations are computed by drawing random words from a sampling distribution \(P_n(w), \forall w \in V\). Typically the number of negative samples \(K\) is much smaller than \(V\) (e.g., \(k \in [5, 20]\)), and hence roughly a \(V/K\) times of speed-up.

Even though negative sampling is an effective approximation technique, as the size of the corpus is typically at the order of billions of words and vocabulary size is at the order of millions (e.g., \(T = 10^9\), and \(V = 10^9\)), training word2vec model often takes tens of hours or even days for some Internet scale applications.

III. WORD2VEC ALGORITHM AND IMPROVEMENTS

In order to solve the optimization problem described in the previous section, Stochastic Gradient Descent (SGD) variants are commonly used. SGD is an iterative algorithm; at each iteration, a single \((w_I, w_O)\) pair is picked where \(w_I\) is a context word and \(w_O\) is a target word or a negative sample. The gradient of the objective function is then calculated w.r.t. the word vectors for \(w_I\) and \(w_O\); and a small change/update is made to these vectors. One of the problems of SGD is that it is inherently challenging to parallelize, i.e., SGD only updates the word vectors of a pair of words at a time, and parallel model updates on multiple threads can result in conflicts if the threads try to update the vectors for the same word.

The original implementation of word2vec by Mikolov et al.\(^1\) uses Hogwild [8] to parallelize SGD. Hogwild is a parallel SGD algorithm that seeks to ignore conflicts between model updates on different threads and allows updates to proceed even in the presence of conflicts. The pseudocode for Hogwild SGD update is shown in Algorithm 1. The algorithm takes in a matrix \(M_{in}^{V \times D}\) that contains the word representations for each input word, and a matrix \(M_{out}^{V \times D}\) for the word representations of each output word. Each word is represented as an array of \(D\) floating point numbers, corresponding to one row of the two matrices. These matrices are updated during the computation. We also take in a specific target word, and a set of \(N\) context words around the target as depicted in Fig. 2. The algorithm iterates over the \(N\) context words in Lines 2-3. The pseudocode only shows a single thread; in Hogwild, the loop in Line 2 is parallelized over threads without any additional change in the code. In the loop at Line 6, we pick either the positive example (the target word in Line 8) or a negative example at random (Line 10). Lines 13-15 compute the gradient of the objective function with respect to the choice of context and positive/negative example. Lines 17–20 perform the update to the entries \(M_{out}[\text{pos/neg example}]\) and \(M_{in}[\text{context}]\).

Algorithm 1: Hogwild SGD implementation of word2vec in one thread.

```
1 Given model parameter Ω = \(\{M_{in}, M_{out}\}\), learning rate α,
2 1 target word \(w_{out}^1\), and N context words \(\{w_{in}^1, w_{in}^2, \ldots, w_{in}^N\}\)
3 for (i = 0; i < N; i++) {
4     \text{input_word = \(w_{in}^i\);}  
5     for (j = 0; j < D; j++) 
6         temp[j] += err * \(M_{in}[^{\text{input_word}}][j]\);
7     \text{temp_word = sample one word from \(V\); label = 0;}  
8     \text{if (k = 0)}{
9         \text{target_word = \(w_{out}^k\); label = 1;}  
10        \text{for (j = 0; j < D; j++) \text{inn +=} \(M_{out}[\text{target_word}][j]\);}  
11        err = label - \(\sigma(\text{inn})\);  
12        \text{for (j = 0; j < D; j++) \text{temp[j] += err *} \(M_{out}[\text{target_word}][j]\);}  
13        \text{// update output matrix}
14        \text{for (j = 0; j < D; j++)} \(M_{out}[\text{target_word}][j]\) += α * temp[j];  
15        \text{// update input matrix}
16        \text{for (j = 0; j < D; j++)} \(M_{in}[\text{input_word}][j]\) += α * temp[j];}
17 }
```

\(^1\)https://code.google.com/archive/p/word2vec/

A. Advantages and Drawbacks of Algorithm 1

Algorithm 1 has a few main advantages: threads do not need to synchronize between updates and can hence proceed independently with minimal instruction overheads. Further, the computation of the gradient is based off the current state of the model visible to the thread at that time. Since all threads update the same shared model, the values read are only as stale as the communication latency between threads, and in practice this does not cause much convergence problems for word2vec.

B. Shared Memory Parallelization

However, the algorithm suffers from two main drawbacks that significantly affect runtimes. First, since multiple threads can update the same cache line containing a specific model entry, there can be significant ping-ponging of cache lines across cores. This leads to high access latencies and significant drop in scalability. Second and perhaps even more importantly, there is a significant amount of locality in the model updates that is not exploited in the Hogwild algorithm. As an example, we can easily see that the same target word is used in the model updates for several context words. By performing a
In order to convert this to a BLAS3 operation, we also need to batch the input context words. Doing so allows us to convert the original dot-product based multiply into a matrix-matrix multiply call (GEMM) as shown on the right side of Fig. 2. At the end of the GEMM, the model updates for all the word vectors for all context words and sample words that are computed need to be written back. Performing matrix-matrix multiplies (GEMMs) rather than dot-products allows us to leverage all the compute capabilities of modern architectures including instruction set features such as multiply-add instructions in the Intel AVX2 instruction set. It also allows us to leverage heavily optimized linear algebra libraries. Note that typical matrix dimensions are not very large, for instance the number of negative samples is only 5–20, and the batch size for the input batches are limited to about 10–20 for convergence reasons. Nevertheless, we find that we get considerable speedups even with this level of reuse over the original word2vec.

1) Multithreading: While Hogwild performs model updates and potentially the inter-core communication that comes with it after each dot product, our BLAS3 scheme above performs a number of dot products as a GEMM (corresponding to multiple input contexts and multiple samples) before performing model updates. We follow a simple “Hogwild” style philosophy for multi-threading across the GEMM calls - we allow for threads to potentially conflict when updating the models at the end of the GEMM operation.

It is important to note that the locality optimization has a secondary but important benefit - we cut down on the total updates to the model. This happens since the GEMM operation performs a reduction (in registers/local cache) to an update to a single entry in the output matrix; while in the Hogwild scheme such updates to the same entry (same context word representation, for instance) happen at distinct periods of time with potential ping-pong traffic happening in between. As we will see in Sec. IV when we present results, this leads to a much better scaling of our approach than the original word2vec.
2) Impact on convergence: We need to pay careful emphasis to convergence while doing these transformations. In contrast to Hogwild that does small partial model updates frequently, the GEMM based approach batches many model updates together and performs less frequent updates. This can result in different multi-threading behavior; specifically it is possible that threads read a more up-to-date model in Hogwild as opposed to the GEMM based scheme. The extent to which this occurs is, of course, dependent on the batch size we use for the inputs. In our experiments, with a batch size of about 10-20, we have not found any significant impact on convergence. One reason for this is that many of the intermediate model updates in Hogwild are to parts of the model that will be updated again in the very near future – for example, updates to the same context word due to multiple sample words occur close by in time. Even if relatively updated models are seen in Hogwild, they are still not the final result.

3) Comparison to BIDMach: The word2vec implementation in BIDMach [9] also uses the previously described shared negative sampling idea. However, the computation in BIDMach is organized in a different way. First, BIDMach separates the handling of the positive examples and negative samples into two steps. For handling positive examples, BIDMach iterates over each word and performs dot products of word vectors considering that word as the target and surrounding words as context. We can think of these operations as a sequence of matrix vector products, each time with a single target and corresponding context words. There is some reuse of context words across matrix vector calls due to the overlap in context between successive target words. However, since computation is not batched into higher level BLAS calls, BIDMach cannot fully exploit this reuse through standard techniques such as register and cache blocking – register and cache state may not be maintained across loop iterations. In a similar way, BIDMach also processes negative samples as a sequence of dot products, and suffers from similar limitations. In contrast, we directly exploit reuse of context words across multiple matrix vector calls due to the overlap in context between successive target words. However, since computation is not batched into higher level BLAS calls, BIDMach cannot fully exploit this reuse through standard techniques such as register and cache blocking – register and cache state may not be maintained across loop iterations. In a similar way, BIDMach also processes negative samples as a sequence of dot products, and suffers from similar limitations.

C. Distributed Memory Parallelization

To scale out the word2vec computation, we also explore different techniques to parallelize word2vec computation across nodes in a compute cluster. Since the individual matrix multiplies are not very large, there is not too much performance that can be gained from distributing these across multiple nodes (a.k.a. model parallelism). Therefore, data parallelism is considered for distributed implementation. In data parallelism with \( N \) computing nodes, the training corpus is equally partitioned into \( N \) shards and the model \( \Omega = \{ M_{in}, M_{out} \} \) is replicated on each computing node; each node then independently processes the data partition it owns and updates its local model, and periodically synchronizes the local model with all the other \( N - 1 \) nodes.

There are two common issues to be addressed in data parallelism: (1) efficient model synchronization over the communication network, and (2) improving the statistical efficiency of large mini-batch SGD. The first issue arises because typical network bandwidths are an order of magnitude lower than CPU memory bandwidths. For example, in commonality cloud computing infrastructures such as AWS the network bandwidths are around 1GB/sec; even in HPC system with FDR infiniband, the network bandwidths are still of the order of 10GB/sec. As the typical size of the model \( \Omega \) is about 2.5GB in our experiments, full model synchronization over 4 computing nodes connected via FDR Infiniband takes about 0.5 seconds, which is too slow to keep up with local model updates. In the case of word2vec, however, not all word vectors are updated at the same frequency as those are proportional to the word unigram frequencies, e.g., the vectors in the model associated with popular words are updated more frequently than those of rare words. We therefore strive to match model update frequency to word frequency, and a sub-model (instead of full-model) synchronization scheme, similar to the one exploited in BIDMach [9], is used.

The second issue arises because as the number of nodes \( N \) increases, conceptually a \( N \) times larger mini-batch is used in SGD update, which affects the statistical efficiency and slows down the rate of convergence. Fortunately, this issue has been studied recently and various techniques are proposed to mitigate the loss of convergence rate. We follow the \( m \)-weighted sample scheme studied in Splash [12] and increase the starting learning rate as the number of nodes increases while explore different learning rate scheduling techniques, such as AdaGrad and RMSProp, to improve convergence rate. From our experiments, we found that while AdaGrad and RMSProp are effective techniques to speed up convergence, they incur large memory consumption since they dedicate a learning rate to each model parameter and need a separate matrixes of the same size as \( \Omega \) to store the per-parameter learning rates. In addition, accessing large memory arrays makes the algorithm memory-bandwidth intensive and slows down the throughput considerably. Instead, we found that a simple learning rate update schedule based on a single learning rate is quite satisfactory, and empirically we note that we just need to reduce the learning rate more aggressively as number of nodes increases. We demonstrate the effectiveness of these techniques in our experiments next.

IV. EXPERIMENTAL ANALYSIS

With the techniques and optimizations discussed above, our optimized word2vec delivers the highest system-performance, measured as throughput, i.e., million words/sec, reported to date on both single node shared memory systems and multi-node distributed clusters, while maintaining predictive-performance measured as accuracy. This section provides detailed analysis of our algorithm on shared memory and distributed memory systems.
A. Experimental Setup

**Hardware:** The majority of our experiments are performed on dual-socket Intel Xeon E5-2697 v4 Broadwell CPUs for shared memory and distributed memory computation. This processor has 36 cores (72 threads including Simultaneous Multi-Threading/SMT) running at 2.3 GHz. Our machine has 128 GB RAM and runs Red Hat Enterprise Linux Server release 6.5. In the distributed setting, all the computing nodes are connected through FDR infiniband.

**Software:** We use custom end-to-end code written in C++ with OpenMP, and compiled with the Intel C++ Compiler version 16.0.2. We use Intel MKL version 11.3.2 and Intel MPI library version 5.1.3 for SGEMM calls and multi-node message passing.

**Training corpora:** We train our word2vec models on three different corpora: (1) a small (text8) dataset of one million words from wikipedia that is widely used for word embedding demos, (2) the recently released one billion word language modeling benchmark [13], and (3) a large collection of 7.2 billion words that we gathered from a variety of data sources: the 2015 Wikipedia dump with 1.6 billion tokens, the WMT14 News Crawl with 1.7 billion tokens, the aforementioned one billion word benchmark, and UMBC webbase corpus with around 3 billion tokens. Different corpora are used in order to verify the generalization performance of our algorithm under different training data statistics. The one billion word benchmark [13] is our main dataset for throughput and predictive performance study since this is the benchmark on which the best known GPU performances were reported.

**Test sets:** The quality of trained models are evaluated on word similarity and word analogy tasks. For word similarity, we use WS-353 [14] which is one of the most popular test datasets used for this purpose. It contains word pairs together with human-assigned similarity judgments. The word representations are evaluated by ranking the pairs according to their cosine similarities, and measuring the Spearman’s rank correlation coefficient with the human judgments. For word analogy, we use the Google analogy dataset [11], which contains 19544 word analogy questions, partitioned into 8869 semantic and 10675 syntactic questions. The semantic questions contain five types of semantic analogies, such as capital cities (Paris;France;Tokyo:?), currency (USA:dollar;India:?) or people (king;queen;man:?). The syntactic questions contain nine types of analogies, such as plural nouns, opposite, or comparative, for example good:better;smart:?. A question is correctly answered only if the algorithm selects the word that is exactly the same as the correct word in the question.

**Code:** We compare the performances of three different implementations of word2vec: the original implementation from Google that is based on Hogwild SGD on shared memory systems (https://code.google.com/archive/p/word2vec/) and BID-Mach (https://github.com/BIDData/BIDMach) which achieves the best known performance of word2vec on Nvidia GPUs, and our optimized implementation on Intel architectures. Our code will be made available for general usage.

**Word2vec parameters:** In the experiments on the one billion word benchmark, we follow the parameter settings of BIDMatch (dim=300, negative samples=5, window=5, sample=1e-4, vocabulary of 1,115,011 words). In this case, the size of the model $\Omega = \{M_{in}, M_{out}\}$ is about 2.5GB. Similar parameter settings are used for the small text8 dataset and the 7.2 billion word collection.

B. Single Node System With Shared Memory

To achieve high performance on modern multi-socket multicore shared memory systems, parallel algorithms need to have strong scalability across cores and sockets. Scaling across cores is challenging for word2vec because more threads creates more inter-core traffic due to cache line conflicts (including false sharing), which prevents it from achieving good scalability and system performance. Scaling across sockets is even more challenging since the same traffic caused by cache line conflicts and false sharing needs to travel across sockets. The high inter-socket communication overhead imposes a major hurdle to achieve good scalability across sockets.

**System-Performance (Throughput):** Fig. 3 shows the system-performance (million words/sec) of our algorithm and the original word2vec, scaling across all cores/threads and sockets of a 36-core dual-socket Intel Broadwell CPU. We use the one billion word benchmark [13] in the experiment. When using only one thread, our optimization achieves 2.6X speedup over the original word2vec. The superior performance of our optimization is due to our parallelization scheme which is more hardware-friendly after converting BLAS1 dot-products to BLAS3 matrix multiplies as described in Sec. III.

---

2http://mattmahoney.net/dc/text8.zip 
3http://www.statmt.org/wmt14/translation-task.html 
4http://ebiquity.umbc.edu/resource/html/id/351

---

Fig. 3: Scaling of the original word2vec and our optimization on all threads of a dual-socket 36-core Intel Xeon E5-2697 v4 CPU; evaluated on the one billion word benchmark [13].
When scaling to multiple threads, our algorithm achieves linear speedup as shown in Fig. 3. This linear scalability is near perfect within a single socket (when number of threads ≤ 36), and the scalability becomes sub-linear when two sockets are involved (when number of threads = 72) in which case cross-socket memory access penalizes the potential linear scaling. In contrast, the original implementation of word2vec scales linearly only until 8 threads and slows down significantly after that. In the end, the original word2vec delivers about 1.6 million words/sec, while our code delivers 5.8 million words/sec or a 3.6X speedup over the original word2vec. The superior performance and scalability highlights the effectiveness of our optimization in reducing unnecessary inter-thread/core communications compared to the original word2vec.

Predictive-Performance (Accuracy): Delivering higher throughput is only meaningful when the trained model reaches similar or better predictive performance. We therefore evaluate the models trained from the original word2vec and our implementation, and report their predictive performances on the word similarity and word analysis tasks in Table I. In order to verify the generalization performance of our techniques, we run the respective codes on three different training corpora as described above.

TABLE I: Predictive performances of the models trained from the original word2vec and our implementation on three different training corpora. All the experiments are performed on a dual-socket 36-core Intel Xeon E5-2697 v4 CPU.

| Corpus             | Vocabulary Size | Word Similarity   | Word Analogy   |
|--------------------|-----------------|-------------------|----------------|
|                    |                 | Original | Our         | Original | Our         |
| 1M-word (text8)    | 71,291          | 63.4     | 66.5 | 17.2 | 18.1         |
| 1B-word benchmark  | 1,115,011       | 64.0     | 64.1 | 32.4 | 32.1         |
| 7.2B-word collection | 1,115,011     | 70.0     | 69.8 | 73.5 | 74.0         |

As can be seen from Table I, our code achieves very similar predictive performance as the original word2vec. This demonstrates that our implementation generalizes well to different corpora and achieves similar (sometimes even slightly better) accuracy compared to the original word2vec.

To examine the robustness of our word2vec thoroughly, we further study the predictive performance under varying data statistics. We again run the original word2vec and our optimization on the one billion word benchmark but with vocabularies of different sizes. For the vocabulary of size $N$, we keep the top $N$ most popular words occurred in the corpus in the vocabulary. These popular words have the most of occurrences in the training corpus, and therefore their updates (and also the conflicts in the “Hogwild”-style SGD) are more frequent than those on rare words. It can been seen from Table II that both the original word2vec and our parallelization achieve very similar predictive performance for all vocabulary sizes, including the most challenging one with a small vocabulary of 50K words.

Overall, these experiments demonstrate that the parallelization scheme and the optimizations we proposed delivers 3X-4X speedup over the original word2vec without loss of predictive performance.

TABLE II: Predictive performances of trained models on one billion word benchmark with vocabularies of different sizes.

| Vocabulary Size | Word Similarity | Word Analogy |
|-----------------|-----------------|--------------|
| Original        | Our             | Original     | Our     |
| 1,115,011       | 64.2            | 32.4         | 32.1     |
| 300,000         | 63.0            | 32.2         | 33.0     |
| 250,000         | 63.1            | 32.2         | 33.0     |
| 100,000         | 55.6            | 32.2         | 31.9     |
| 50,000          | 49.7            | 30.1         | 29.9     |

Comparison to state-of-the-arts: After demonstrating the superior performances of our optimization, we now perform detailed comparison to the state-of-the-arts, including the original word2vec from Google and BIDMach. Since all the implementations achieve similar accuracy, we focus on the throughput in the comparison. Improving throughput (while maintaining accuracy) is always important since it democratizes the large word2vec models by lowering training costs. Thus, extensive studies have been focused on improving throughput. The best known performance reported currently in the literature is from BIDMach on the one billion word benchmark running on Nvidia GPUs [9]. We therefore run our experiments on the same benchmark using the same parameter setting as that of BIDMach. Moreover, to evaluate the generalization of our techniques, we also run our experiments on two different Intel architectures.

Table III shows the detailed comparisons. On both Intel architectures, BIDMach and our optimization outperform the original word2vec: typically BIDMach delivers 1.6X speedup over the original word2vec while our optimization delivers 2.8X-3.6X speedup. In addition, our performance on Intel Broadwell (5.8 million words/sec) outperforms BIDMach’s performance on Nvidia K40 (4.2 million words/sec). The best known shared memory performance was reported on

| Processor       | Code       | Words/Sec |
|-----------------|------------|-----------|
| Intel HSW (Xeon E5-2690 v3) | Original | 1.5M      |
| Intel HSW (Xeon E5-2680 v3) | BIDMach    | 2.4M      |
| Intel HSW (Xeon E5-2680 v3) | Our        | 4.2M      |
| Intel BDW (Xeon E5-2697 v4) | Original | 1.6M      |
| Intel BDW (Xeon E5-2697 v4) | BIDMach    | 2.5M      |
| Nvidia K40      | BIDMach    | 4.2M      |
| Nvidia GeForce Titan-X | BIDMach | 5.8M     |

*Data from [9].
Nvidia GeForce Titan-X (8.5 million words/sec) by BIDMach which is 1.5X faster than our performance on Intel Broadwell. However, in terms of compute efficiency, BIDMach on Nvidia Titan-X is much lower than our code on Intel Broadwell since the former has 3X peak flops of the latter, indicating that BIDMach’s efficiency on Nvidia Titan-X is only half of ours on Intel Broadwell. This is likely due to the parallelization scheme of BIDMach, which cannot efficiently use all computational resources, as we discussed in Sec. III. Moreover, we will report our performance on the latest Intel Knights Landing processor in the final version of this paper if this paper gets accepted.

C. Distributed Multi-node Clusters

Scalability and performance on multi-node distributed systems are at least as important as, if not more important than, those on single node systems. This is because typical large scale machine learning applications are compute intensive and require hours, days, even weeks of training time. In particular, even with our optimized word2vec (or BIDMach on Titan-X GPU) it still takes tens of hours or even days to train on some of the largest datasets, such as the 100 billion word news articles from Google. Thus, strong scalability and predictive performance on distributed multi-node systems are critical in practice.

Next we study the scalability and predictive performance of our techniques on distributed multi-node systems. The experiments with our optimized word2vec are performed on multi-node clusters, containing dual-socket 36-core Intel Broadwell E5-2697 v4 nodes connected via FDR Infiniband. Fig. 4 shows the scalability of our distributed word2vec as number of nodes increases, while Table IV reports the corresponding predictive performances. Again, good scalability is only meaningful when similar or better accuracy is achieved. We therefore provide the accuracy of the original word2vec as the baseline in Table IV.

As can been seen from Fig. 4 and Table IV, our distributed word2vec achieves near linear scaling until 16 nodes while maintains a comparable accuracy to that of the original word2vec. To achieve this linear scaling on the throughput, we increase the learning rate and the model synchronization frequency slightly to mitigate the loss of convergence rate as the number of nodes increases. When number of nodes increases to 32, we need to further increase model synchronization frequency to maintain a good predictive accuracy. However, the increment of model synchronization frequency takes a toll on the scalability, and leads to a sub-linear scaling at 32 nodes. Despite of this, our distributed word2vec delivers about 110 million words/sec with a small 1% accuracy loss. To the best of our knowledge, this is the best throughput reported so far on this benchmark.

V. Conclusion

A high performance parallel word2vec algorithm in shared memory and distributed memory systems is proposed. It combines the idea of Hogwild and minibatching to convert BLAS1 vector-vector operations that are memory-bandwidth intensive to BLAS3 matrix multiply operations for speed. As a consequence, model updates become bigger but less-frequent as compared to the original word2vec implementation. We also explore different techniques for distributed word2vec such as sub-model synchronization and learning rate scheduling. These techniques dramatically reduce network communication 

![Fig. 4: Scaling of our distributed word2vec on multiple Intel Broadwell nodes.](image)

| #Nodes | Word Similarity | Word Analogy |
|--------|-----------------|--------------|
| Original (1) | 64.0 | 32.4 |
| Our (1)   | 64.1 | 32.1 |
| Our (2)   | 64.1 | 32.3 |
| Our (4)   | 63.0 | 32.0 |
| Our (8)   | 63.8 | 32.1 |
| Our (16)  | 62.8 | 31.6 |
| Our (32)  | 63.2 | 31.1 |

Intel Broadwell nodes, the best performance reported so far on this benchmark.

| Systems          | Node Count | Code   | Words/Sec |
|------------------|------------|--------|-----------|
| Nvidia Titan-X   | 4 nodes    | BIDMach| 20M\(^1\) |
| Dual socket Xeon E5-2697 v4 | 4 nodes    | Our    | 20M       |
| Dual socket Xeon E5-2697 v4 | 32 nodes   | Our    | 110M\(^1\) |

\(^1\)Data from [9].
and keep the model synchronization effectively while maintain the rate of convergence when number of nodes increases. We demonstrate the throughput and predictive performance of our algorithm comparing to the state-of-the-arts implementations: the original word2vec and BIDMach on three different corpora. We achieve near linear scalability across cores and nodes, and process hundreds of millions of words per second.

As for future work, our plans include asynchronous model update similar to parameter sever [15], more efficient sub-model synchronization strategy as well as improving the rate of convergence of the distributed word2vec implementation.

REFERENCES

[1] C. D. Manning and H. Schütze, Foundations of Statistical Natural Language Processing. Cambridge, MA, USA: MIT Press, 1999.
[2] R. Collobert and J. Weston, “A unified architecture for natural language processing: deep neural networks with multitask learning,” in Proceedings of the 25th international conference on Machine learning, 2008, pp. 160–167.
[3] X. Glorot, A. Bordes, and Y. Bengio, “A unified architecture for natural language processing: deep neural networks with multitask learning,” in Proceedings of the 25th international conference on Machine learning, 2011, pp. 513–520.
[4] P. D. Turney, “Distributional semantics beyond words: Supervised learning of analogy and paraphrase,” in Transactions of the Association for Computational Linguistics (TACL), 2013, pp. 353–366.
[5] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in Advances in Neural Information Processing Systems 26, 2013, pp. 3111–3119.
[6] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” in Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.
[7] J. Weston, S. Chopra, and A. Bordes, “Memory networks,” in International Conference on Learning Representations (ICLR), 2015.
[8] F. Niu, B. Recht, C. Re, and S. J. Wright, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in Advances in Neural Information Processing Systems, 2011, pp. 693–701.
[9] J. Canny, H. Zhao, Y. Chen, B. Jaros, and J. Mao, “Machine learning at the limit,” in IEEE International Conference on Big Data, 2015.
[10] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” Proceedings of Workshop at ICLR, 2013.
[11] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin, “Placing search in context: The concept revisited,” ACM Transactions on Information Systems, vol. 20, pp. 116–131, 2002.