Proceedings Paper:
Moon, G.E., Newman-Griffis, D. orcid.org/0000-0002-0473-4226, Kim, J. et al. (3 more authors) (2020) Parallel data-local training for optimizing Word2Vec embeddings for word and graph embeddings. In: 2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC). 2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), 18 Nov 2019, Denver, CO, USA. IEEE , pp. 44-55. ISBN 978-1-7281-5986-7

https://doi.org/10.1109/mlhpc49564.2019.00010

Parallel Data-Local Training for Optimizing Word2Vec Embeddings for Word and Graph Embeddings

Gordon E. Moon
Computer Science and Engineering
The Ohio State University
Columbus, OH, U.S.A.
moon.310@osu.edu

Denis Newman-Griffis
Computer Science and Engineering
The Ohio State University
Columbus, OH, U.S.A.
newman-griffis.1@osu.edu

Jinsung Kim
School of Computing
University of Utah
Salt Lake City, UT, U.S.A.
u6027797@utah.edu

Aravind Sukumaran-Rajam
Computer Science and Engineering
The Ohio State University
Columbus, OH, U.S.A.
sukumaranrajam.1@osu.edu

Eric Fosler-Lussier
Computer Science and Engineering
The Ohio State University
Columbus, OH, U.S.A.
fosler-lussier.1@osu.edu

P. Sadayappan
School of Computing
University of Utah
Salt Lake City, UT, U.S.A.
saday@cs.utah.edu

***Abstract*—The Word2Vec model is a neural network-based unsupervised word embedding technique widely used in applications such as natural language processing, bioinformatics and graph mining. As Word2Vec repeatedly performs Stochastic Gradient Descent (SGD) to minimize the objective function, it is very compute-intensive. However, existing methods for parallelizing Word2Vec are not optimized enough for data locality to achieve high performance. In this paper, we develop a parallel data-locality-enhanced Word2Vec algorithm based on Skip-gram with a novel negative sampling method that decouples loss calculation with positive and negative samples; this allows us to efficiently reformulate matrix-matrix operations for the negative samples over the sentence. Experimental results demonstrate our parallel implementations on multi-core CPUs and GPUs achieve significant performance improvement over the existing state-of-the-art parallel Word2Vec implementations while maintaining evaluation quality. We also show the utility of our Word2Vec implementation within the Node2Vec algorithm which accelerates embedding learning for large graphs.**

***Index Terms*—Parallel Machine Learning, Unsupervised Learning, Learning Latent Representations, Parallel Word2Vec, Node2Vec, Word Embedding, Graph Embedding**

I. Introduction

The Word2Vec model proposed by Mikolov et al. [1], [2] belongs to the family of neural network-based static word embedding techniques that generate a dense vector in a low-dimensional embedding space for each word in a fixed vocabulary. The embeddings generated by Word2Vec can be utilized as key features in a wide range of applications such as natural language processing [3], [4], bioinformatics [5], [6] and graph mining [7], [8]. While embeddings are often trained once on a large corpus and re-used for a variety of applications, several results have demonstrated that for tasks in specific domains, such as biomedicine, embeddings trained *de novo* on domain-specific corpora outperform general-purpose embeddings [9]. Additionally, recent work has shown that training many sets of embeddings on specific sub-corpora can be used to model language change over time [10], even for specific tasks like tracking armed conflicts [11]. The time cost to train a large number of embedding models limits the expanded use of embeddings for detailed analysis of multiple datasets. In this paper, we present improvements to Word2Vec training that significantly reduce the time required to train high-quality embeddings.

The main focus of this paper is the adaptation of the Word2Vec algorithm with a view towards reducing the amount of data movement from/to memory. Technology trends have made the cost of data movement increasingly dominant, both in terms of energy and time, over the cost of performing arithmetic operations in computer systems. However, the design and implementation of new algorithms in machine learning has been largely driven by a focus on the computational complexity. The inner core computation of the Word2Vec algorithm (explained in great detail later) involves a large number of dot-product operations between the embedding vectors representing different words in the corpus. The dot-product computation is inherently memory-bandwidth limited because only two floating point operations are performed per pair of data elements read from memory. Since the computational peak performance of all current/emerging processors (multi-core CPUs, GPUs, FPGAs, etc.) greatly exceeds the peak memory bandwidth in words/second (often by a factor between ten and hundred), any algorithm that performs a large number of dot-products is inherently performance-handicapped, unless significant re-use of the processed vectors in caches can be achieved.

The Skip-gram based Word2Vec algorithm (described in detail later) processes words within small contiguous windows in the text, using dot-products of the center "focus" word with other words in the window in the process of trying to move the vector embedding of the word closer to that of the neighboring ones in the window (*Attraction*). Dot-products with a large number of randomly selected words in the sentence are also performed with the words in the window, seeking to move their vector embeddings further from them since they are not found co-located in the corpus (*Repulsion*). The random access to words in the latter negative sampling process leads to very

high data movement without any prospect of reuse of those randomly fetched vectors. We develop an adaptation to the way the negative sampling is performed that enables much higher data reuse for the fetched vectors. This is done by separating out the computations that seek to align a word closer to its neighbors in the windows from the negative-sampling computations, which are performed on mini-batches of words at a time, using a common set of randomly chosen words to move away from.

We develop efficient multi-core and GPU implementations of the adapted Word2Vec algorithm and perform extensive comparative evaluation of the new algorithms with a number of state-of-the-art implementations of Word2Vec on both platforms. We also show the utility of the new Word2Vec implementation within the Node2Vec algorithm, which accelerates embedding learning for large graphs. We demonstrate significant reduction of data volume from/to main memory and improved performance over the current state-of-the-art. We also conduct extensive evaluation on the quality of the produced embeddings, for a number of application contexts. Overall, we find that we are able to achieve high performance with quality of results that are comparable or better than baselines.

II. SG-NS based Word2Vec Algorithm

The basic intuition behind the Word2Vec model proposed by Mikolov et al. [1], [2] is that similar words tend to occur in similar contexts. Word2Vec is trained as a language model, in which the probability of a word in a text corpus depends on its surrounding context words. Word2Vec has two primary variants: (1) Continuous Bag-of-Words (CBOW), which predicts a center word using the average of the words in a fixed context window on either side, and (2) Skip-Gram (SG), which models pairwise probabilities of the center word with each of its context words individually. These two algorithms therefore result in different numbers of parameter updates: CBOW only updates the center word once for a given context window, while SG updates the center word once for each context word. In practice the algorithms yield roughly equivalent performance ([12] found better results with CBOW; [2], [13] with SG); we follow prior work in focusing on the SG algorithm, which can (unlike CBOW) be interpreted as implicit factorization of the word co-occurrence matrix [14]. As defined by [1], the SG model seeks to maximize the average log probability $J(\theta)$ given a sequence of N word tokens in the text corpus.

$$J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{-C\le j\le C, j\neq 0}\log p(w_{n+j}|w_n) \qquad (1)$$

where C is the window size around center word w_n. The probability of $p(w_{n+j}|w_n)$ is defined with the softmax function as:

$$p(w_{n+j}|w_n) = \frac{\exp(<\vec{\mathbf{w}}_{n+j}^{out}, \vec{\mathbf{w}}_n^{in}>)}{\sum_{v=1}^{V}\exp(<\vec{\mathbf{w}}_v^{out}, \vec{\mathbf{w}}_n^{in}>)} \qquad (2)$$

where V is the size of the fixed vocabulary, $\vec{\mathbf{w}}_n^{in}$ and $\vec{\mathbf{w}}_v^{out}$ denote the vector from word w_n in the W_{in} matrix and the vector for word w_v in the W_{out} matrix, respectively. $< \vec{\mathbf{w}}_v^{out}, \vec{\mathbf{w}}_n^{in} >$ computes the inner product of word vectors $\vec{\mathbf{w}}_v^{out}$ and $\vec{\mathbf{w}}_n^{in}$.

SG and CBOW can be trained using one of two different objectives. Given a center word w_n and an observed target word w_{n+j}, calculating the full softmax equation in Equation 2 for every word update becomes rapidly intractable as the vocabulary size V increases. Instead of training probabilistically with the full softmax, the negative sampling (NS) method randomly chooses a small number of words in the vocabulary as negative targets, and trains using a log-bilinear objective derived from noise-contrastive estimation. The word vectors are updated based on only the selected negative target words and a positive target word w_{n+j}, considerably reducing the number of computations while maintaining a good quality of solution [2]. Alternatively, the hierarchical softmax (HS) objective originally proposed by Morin and Bengio [15], which approximates the full softmax function, can also be used to reduce the computational complexity of probabilistic training. Mikolov et al. [2] use the binary Huffman tree structure for hierarchical softmax to compute the probability distribution of a given word (root node) along with the path from root to vocabulary size of leaf nodes. Each path in the tree structure represents the relative probability to its child node. By the nature of the binary Huffman tree, the path between the root node and a leaf node is shorter when the corresponding words are frequently used together in the corpus. This characteristic of hierarchical softmax makes neural network based models more efficient in terms of both elapsed time for training and accuracy. Typically, SG and CBOW are paired with each of the negative sampling (SG-NS) and hierarchical softmax (CBOW-HS), respectively. Hereafter, we concentrate on the skip-gram based Word2Vec with negative sampling (SG-NS) algorithm.

A. Node2Vec

The Node2Vec algorithm [7] is intended to learn vector-based representations of nodes in a graph, such that similar nodes (nodes in similar subgraph structures or nearby nodes) have similar vectors. Node2Vec consists of sampling R random walks of length L starting from each node in the graph; each walk becomes a sentence for Word2Vec training. So for V unique nodes, Node2Vec generates a corpus of $V \times R$ sentences of L tokens (node IDs) each. These walks are then run through off-the-shelf SG-NS based Word2Vec, with each node and its neighbor nodes in the walk corresponding to word tokens w_n and w_{n+j} in Equation 1, respectively. Thus, each node in the pre-generated graph dataset becomes a word in the corpus. The effectiveness of the node embedding outputs, W_{in} and W_{out}, are then directly evaluated using applications such as node classification and link prediction [7], [8], [16]. Our contribution is to achieve a significant speedup on the embedding learning part of the Node2Vec algorithm by transforming the original Word2Vec into our new parallel Word2Vec algorithm. In fact, the Word2Vec embedding training portion of running Node2Vec on the BlogCatalog dataset accounts for *training time of Word2Vec within the Node2Vec / total training time of Node2Vec* = 72.635360(s) / 81.505412(s) $\approx$ 89.12% of the total run time.

III. Related Work

A. Parallelization of Word2Vec Embeddings

Several efficient data-locality-enhanced schemes have recently been proposed for Word2Vec algorithms because

the original Word2Vec algorithm imposes massive computations. As shown in Table I, previous studies at parallelizing Word2Vec can be grouped by the types of machine, platform, and algorithm used in their implementations.

The original Word2Vec implementation released by Mikolov et al. [2] adopts HOGWILD! [17], a parallelization scheme where different word pairs are simultaneously processed across multiple threads. HOGWILD! partially overcomes the limitation of Stochastic Gradient Descent (SGD) optimization, which is inherently challenging to parallelize. However, adopting HOGWILD! in the Word2Vec algorithm has the inevitable limitation that it may lead to a race condition between different threads when updating the same word vector in the input and output matrices at the same time. In a shared-memory environment, Ji et al. [18] proposed a parallel Word2Vec (pWord2Vec) model that maximizes reuse of data structures by sharing negative samples within the same context window. The proposed scheme changes the original Level 1 BLAS operations into Level 3 BLAS operations; thereby, the number of updates and communication cost between threads are considerably reduced. The main difference between pWord2Vec and our scheme is the maximum number of input words (columns in yellow colored matrix M_{in} in Figure 4) sharing the same negative samples. In pWord2Vec, the maximum number is restricted to the total number of input words in each context window, which is $2 \times window\ size + 1$. However, in our approach, the number of columns in matrix M_{in} is maximized using mini-batch size of all words within each sentence, regardless of context window, to share the same negative samples. Vuurens et al. [19] introduced an efficient caching strategy for updating vectors based on the SG-HS algorithm. They maintained local copies of the most frequently used inner nodes in the Huffman tree structure to maximize data reuse in cache and reduce the number of memory conflicts. Recently, Rengasamy et al. [21] introduced a context combining approach (pSGNScc) to further optimize the data reuse in pWord2Vec. In pSGNScc, multiple correlated context windows share not only negative samples but also positive samples. Given the words in the current context window, pSGNScc utilizes a pre-generated inverse index table to find related windows according to the word occurrences in the entire corpus. In their experiments, pSGNScc achieved 1.28X speedup compared to pWord2Vec. In Ordentlich et al. [22], a distributed SG-NS based Word2Vec algorithm was proposed in order to reduce the high training latency and network bandwidth with large-scale datasets. On top of a Hadoop system, multiple servers learn the distributed word vectors to achieve the higher throughput in parallel. In the GPU platform, Simonton and Alaghband [20] developed both SG-HS and SG-NS based Word2Vec algorithms using shared memory registers and in-warp shuffle operations on GPUs. Within a thread block, the use of shared memory registers significantly reduces the data accessing time compared to the global memory access. Based on roofline design, Canny et al. [24] developed a system called BIDMach to improve the performance of different machine learning algorithms. As it was reported that their SG-NS based GPU implementation suffers from the quality of word embeddings (see Table 2 in [20]), their GPU implementation was not included as a competing model. Bae and Yi [23] implemented four variants of Word2Vec model using GPUs. Since the computations of nested loops which iterate over the number of hidden units are dominant in the original Word2Vec algorithm, the number of threads used in their GPU implementation is same as the number hidden units. One of their variants, SG-NS based GPU implementation, was used as a baseline for our new GPU implementation.

TABLE I: Previous studies on parallelization of Word2Vec. Context types – Skip-gram (SG) and Continuous Bag-of-Words (CBOW). Objective types – Negative Sampling (NS) and Hierarchical Softmax (HS)

Author	Machine	Platform	Algorithm
Mikolov et al. [2]	CPUs	Shared-memory	CBOW-NS, CBOW-HS, SG-NS, SG-HS
Ji et al. [18]	CPUs	Shared-memory	SG-NS
Vuurens et al. [19]	CPUs	Shared-memory	SG-HS
Simonton and Alaghband [20]	CPUs	Shared-memory	SG-NS, SG-HS
Rengasamy et al. [21]	CPUs	Shared-memory	SG-NS
Ji et al. [18]	CPUs	Distributed-memory	SG-NS
Ordentlich et al. [22]	CPUs	Distributed-memory	SG-NS
Simonton and Alaghband [20]	GPUs	Shared-memory	SG-NS, SG-HS
Bae and Yi [23]	GPUs	Shared-memory	CBOW-NS, CBOW-HS, SG-NS, SG-HS
Canny et al. [24]	GPUs	Shared-memory	SG-NS, SG-HS

B. Graph Embeddings

Recently, several graph embedding algorithms based on random walk sampling have been developed by utilizing Skip-gram based Word2Vec model to find a low-dimensional latent representation for each node on the graph [7], [8]. It is assumed that there exist similarities between the nodes connected to each other on the random walks. DeepWalk proposed by Perozzi et al. [8] samples random walks over the graph and maps them into the Skip-gram based Word2Vec model for training. To generate node embeddings, they used SG-HS algorithm instead of SG-NS algorithm used in the Node2Vec [7]. More recently, Tang et al. [16] proposed a new edge sampling-based graph embedding algorithm called LINE which incorporates not only the first-order proximity but also second-order proximity between the nodes in the graph.

IV. SG-NS BASED WORD2VEC ALGORITHM

In this section, we describe the original SG-NS based Word2Vec algorithm [2] and main factors that limits the performance.

$$\log \sigma(<\vec{\mathbf{w}}^{out}_{n+j}, \vec{\mathbf{w}}^{in}_{n}>) + \sum_{t=1}^{T} \log \sigma(-<\vec{\mathbf{w}}^{out}_{t}, \vec{\mathbf{w}}^{in}_{n}>) \quad (3)$$

In the SG-NS based Word2Vec model proposed by Mikolov et al. [2], given an input word w_n, a positive target word w_{n+j}, and randomly sampled negative target words $w_{1:T}$, the log probability $p(w_{n+j}|w_n)$ is replaced by the loss calculation in Equation 3, where T is the number of negative samples (targets) and $\sigma(x)$ denotes the sigmoid logistic function defined as $\sigma(x) = 1 \ / \ (1+\exp(-x))$. This term is maximized over every (w_n, w_{n+j}) pair in the corpus. Instead of using vocabulary size of V words as negative targets, randomly selecting only T words (usually $5 \leq T \leq 20$) as negative targets considerably reduces the computational complexity while maintaining good quality. Algorithm 1 shows pseudo-code for the original SG-NS based word2vec implementation [2]. It uses context words as inputs (line 11) and uses the word at the center of the context window as target (line 15), along with the negative

Algorithm 1 SG-NS based Word2Vec algorithm

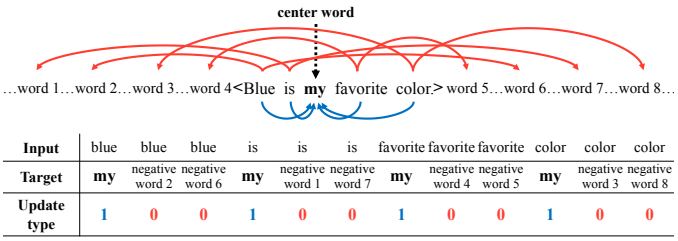
Input: $corpus$: S sentences and a sequence of L word tokens in each sentence, V: the number of unique words, K: the number of hidden units, C: window size, T: the number of negative samples, α: learning rate, W_{in}: $(V \times K)$ input embedding matrix, W_{out}: $(V \times K)$ output embedding matrix
Output: W_{in}: $(V \times K)$ input embedding matrix

```
1: Initialize W_in and W_out with random numbers
2: repeat
3:   for sid = 0 to S − 1 do
4:     L ← number of word tokens in sentence sid
5:     // Update W_out and W_in with both positive and negative samples
6:     for i = 0 to L − 1 do
7:       center_word ← corpus[sid][i]
8:       C_rand ← random_uniform() % C
9:       for j = C_rand to (2 × C − C_rand) do
10:        if j != C then
11:          input ← corpus[sid][i−C+j]
12:          Initialize temp[0:K−1] to 0
13:          for t = 0 to T do
14:            if t == 0 then
15:              target ← center_word, label ← 1
16:            else
17:              target ← random_uniform() % V, label ← 0
18:            end if
19:            sum ← 0
20:            for k = 0 to K − 1 do
21:              sum ← sum + W_in[input][k] × W_out[target][k]
22:            end for
23:            grad ← (label − sigmoid(sum)) × α
24:            for k = 0 to K − 1 do
25:              temp[k] ← temp[k] + grad × W_out[target][k]
26:              W_out[target][k] ← W_out[target][k] + grad × W_in[input][k]
27:            end for
28:          end for
29:          for k = 0 to K − 1 do
30:            W_in[input][k] ← W_in[input][k] + temp[k]
31:          end for
32:        end if
33:      end for
34:    end for
35:  end for
36: until convergence
```

samples (line 17)[1]. The input word vectors are pulled from the W_{in} embedding matrix that is kept after training, and the target word vectors are pulled from W_{out} embedding matrix, which is discarded after training. An input vector and a target vector are multiplied in lines 20-22, in order to compute the gradient in line 23. While training each epoch, based on this gradient, W_{out} and W_{in} matrices are updated (lines 24-27 and lines 29-31 for W_{out} and W_{in}, respectively).

Figure 1 depicts an example of sequential updates for SG-NS based Word2Vec with the sentence "blue is my favorite color," where "my" is the center word of the context window. When the center word "my" is used as a positive target word, its surrounding words "blue", "is", "favorite", and "color" are its input words. Then, for each pair of the center word "my" and one of its input words, two negative target words are randomly chosen to be negative samples (e.g., "word 2" and "word 6" are selected as negative targets where "blue" is the input word). Hereafter, an update with the center word (positive target word) and an input word is called an ***Attraction*** update (blue arrow in Figures 1 and 2), and an update with

[1] [1] define skip-gram using context words as targets and center words as input; however, the reference Word2Vec implementation operates as we have described. This difference only changes the order of updates over the full sequence; in our experiments, Word2Vec had equivalent performance using both definitions of skip-gram.



Input	blue	blue	blue	is	is	is	favorite	favorite	favorite	color	color	color
Target	my	negative word 2	negative word 6	my	negative word 1	negative word 7	my	negative word 4	negative word 5	my	negative word 3	negative word 8
Update type	1	0	0	1	0	0	1	0	0	1	0	0

Fig. 1: An example of updates in the original Word2Vec for the sentence "blue is my favorite color," where "my" is the current center word, and window size and the number of negative samples are both 2. The update types of "1" and "0" indicate *Attraction* and *Repulsion* updates, respectively.

the negative target word and an input word is called ***Repulsion*** update (red arrow in Figures 1 and 2).

A. Data Movement Analysis for SG-NS Based Word2Vec

$$S(L(1 + 2(C - C_{rand})(1 + 4K + 8K(T + 1)))) \quad (4)$$

$$S(L(2(C - C_{rand})(1 + 4K + 8KT))) \quad (5)$$

To identify the data movement cost of the original SG-NS based Word2Vec algorithm, we individually analyzed each line in Algorithm 1. The outer loop in line 3 iterates over all the S sentences in the corpus, and the loop in line 6 iterates over all the L word tokens in each sentence. Each loop in line 6 reads a center word (positive target word) from $corpus$ (1 read). According to the randomly selected context window size, the loop in line 9 iterates over $2C - 2C_{rand}$ surrounding words. Each surrounding input word is read from $corpus$ (1 read), and K size of $temp$ array is initialized by zero (K writes). Then the loop in line 13 calculates vector updates by iterating over one positive and T negative samples ($T + 1$ iterations). After selecting a negative sample, the loop in line 20 multiplies the current input word vector and target word vector ($2K$ reads). The gradient is then computed in line 23. The loop in 24 updates W_{out} by accessing $temp$, W_{out} and W_{in} arrays ($4K$ reads and $2K$ writes). After the completion of the update calculation loop, the loop in line 29 applies the updates to W_{in} ($2K$ reads and K writes). Overall, the total data movement required for the original SG-NS based algorithm is shown in Equation 4. As shown in Equation 5, the data movement cost for only *Repulsion* updates can be simply obtained by excluding the data movement of line 7 and subtracting one iteration of the loop in line 13 from the total data movement. It is evident that the main bottleneck of SG-NS based Word2Vec algorithm is *Repulsion* updates. More specifically, the data movement overhead is closely associated with the several vector-vector multiplications within the loop in line 9. In the next section, we present a high-level overview and details of our new parallel Word2Vec algorithm called **P**arallel decoupled **A**ttraction-**R**epulsion based **Word2Vec** (**PAR-Word2Vec**) based on Skip-gram with a novel negative sampling method. In light of our main goal which is to enhance the algorithm in regards to data locality, our approach significantly alleviates data movement overhead involving *Repulsion* updates. We also compare the

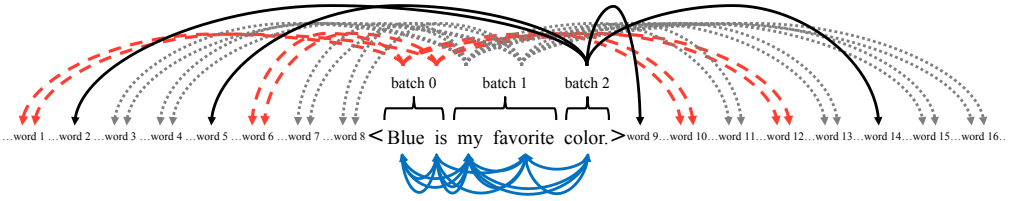
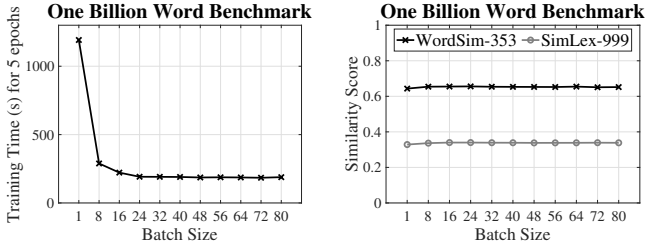


Fig. 2: An example of full *Attraction* and *Repulsion* updates in the PAR-Word2Vec. In the sentence "blue is my favorite color," where the mini-batch size is set to 2, the input words in each mini-batch share the same negative target words. As marked by same colored arrows (red, gray and black), the words repelled by each mini-batch are shared negative target words (e.g., "word 1", "word 6", "word 10" and "word 12" are the shared negative targets for batch 0).



(a) Time vs. Batch size (b) Quality vs. Batch size

Fig. 3: Comparison of averaged training time in seconds and convergence for 5 executions across different mini-batch sizes on PAR-Word2Vec-cpu on the One Billion Word Benchmark dataset, where $K = 128$.

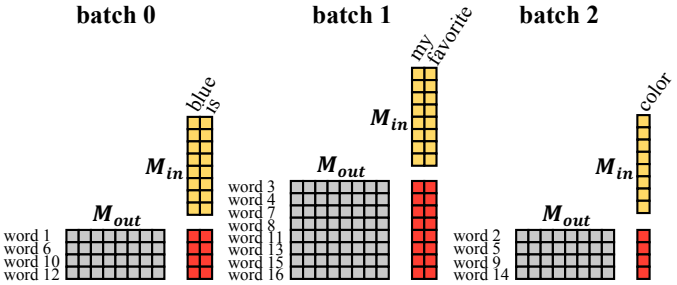


Fig. 4: Decoupled full *Repulsion* updates required in the PAR-Word2Vec with the sentence "blue is my favorite color". In the decoupled *Repulsion* phase, the number of shared negative samples for the mini-batch (i.e., the number of rows in gray colored matrix) is determined by the first word's position in the mini-batch over the sentence. The mini-batches located at the start or end of the sentence (e.g., batch 0 or batch 2) have a less amount of shared negative samples, since the context window size of the input words contained in these mini-batches is smaller than that of the middle mini-batch (e.g., batch 1).

data movements required for our PAR-Word2Vec with original SG-NS based Word2Vec algorithms.

V. PAR-Word2Vec Algorithm

In the original Word2Vec algorithm, majority of training time is spent on the process associated with negative sampling. Our main contribution is to improve the performance of negative sampling method by increasing data reuse. In order to use much larger size of matrix with negative sampling, we first decouple the full loss calculation in Equation 3 with positive samples, $\log \sigma(< \vec{\mathbf{w}}^{out}_{n+j}, \vec{\mathbf{w}}^{in}_{n} >)$, and negative samples, $\sum_{t=1}^{T} \log \sigma(- < \vec{\mathbf{w}}^{out}_{t}, \vec{\mathbf{w}}^{in}_{n} >)$, within each sentence. Thereafter, all of the *Attraction* updates required for each sentence are batched, and then followed by a batch of *Repulsion* updates. After the completion of *Attraction* updates, all words included in each sentence are divided into multiple mini-batches, and the words in the same mini-batch share the randomly chosen words for *Repulsion* updates. For example, in Figure 2, two input words in the same mini-batch (e.g., "blue" and "is" in batch 0) share the negative target words (e.g., "word 1", "word 6", "word 10" and "word 12"), when the mini-batch size is set to 2. If the mini-batch size is larger than or equals to the number of words in the sentence, there is only one mini-batch and all the words within the sentence share the same negative samples.

It is important to determine an appropriate mini-batch size to reduce training time without overly affecting the convergence rate. To analyze the impact of different mini-batch sizes on both convergence rate and training time, we conducted an empirical evaluation with the large text dataset. As depicted in Figure 3a and Figure 3b, increasing the mini-batch size significantly reduces training time because of lower computational complexity without any loss of model quality. Furthermore, in order to perform the *Repulsion* phase as similar to the original SG-NS based algorithm, a different number of shared negative samples is drawn for each mini-batch according to the first word's sentence position in the mini-batch.

Figure 4 illustrates the full set of operations required for decoupled *Repulsion* phases in PAR-Word2Vec. We maintain the *Attraction* phase as the original SG-NS algorithm, since the *Attraction* phase demands an extremely small computation compared to the *Repulsion* phase. Based on the mini-batch processing, however, the *Repulsion* phase is reformulated with matrix multiplications, as shown in Figure 4. For each mini-batch, its shared negative target vectors pulled from the W_{out} matrix can be combined to form a temporary target matrix M_{out}. Likewise, input vectors pulled from W_{in} matrix are combined to generate a temporary input matrix M_{in} for the current mini-batch. Then an efficient matrix multiplication of two matrices M_{out} and M_{in} is performed for computing the gradients. Given the result matrix of gradients, two additional matrix multiplications compute the update values for corresponding input and target vectors which will be accumulated to W_{out} and W_{in} matrices.

Algorithm 2 Parallel CPU implementation (**PAR-Word2Vec-cpu**)

Input: $corpus$: S sentences and a sequence of L word tokens in each sentence, V: the number of unique words, K: the number of hidden units, C: window size, T: the number of negative samples, B: mini-batch size, H: the number of shared negative samples for current batch, α: learning rate, W_{in}: $(V \times K)$ input matrix, W_{out}: $(V \times K)$ output matrix, M_{in}: $(B \times K)$ matrix, M_{out}: $(H \times K)$ matrix, M_{grad}: $(H \times B)$ matrix, M_{in_up}: $(B \times K)$ update matrix, M_{out_up}: $(H \times K)$ update matrix, $shared\_ns$: H size of array
Output: updated W_{in}: $(V \times K)$ input embedding matrix

1: Initialize W_{in} and W_{out} with random numbers
2: $numT \leftarrow$ number of threads
3: **#pragma omp parallel num_threads**($numT$)
4: Distribute S sentences of $corpus$ into $numT$ threads
5: tId $\leftarrow$ thread id
6: $S_{pt} \leftarrow \frac{S}{numT}$
7: **repeat**
8: **for** sid = tId $\times$ S_{pt} **to** (tId + 1) $\times$ S_{pt} $-$ 1 **do**
9: $L \leftarrow$ number of word tokens in sentence sid
10: W_{in}, $W_{out} \leftarrow$ **Attraction**()
11: W_{in}, $W_{out} \leftarrow$ **Repulsion**()
12: **end for**
13: **until** convergence

Algorithm 3 Attraction() phase on **PAR-Word2Vec-cpu**

1: **// Update W_{out} and W_{in} with only positive samples**
2: label $\leftarrow$ 1
3: **for** i = 0 **to** L $-$ 1 **do**
4: center_word $\leftarrow$ $corpus$[sid][i]
5: $C_{rand} \leftarrow random\_uniform()$ % C
6: **for** j = C_{rand} **to** (2 $\times$ C $-$ C_{rand}) **do**
7: **if** j ! = C **then**
8: input $\leftarrow$ $corpus$[sid][i$-C$+j]
9: target $\leftarrow$ center_word
10: sum $\leftarrow$ 0
11: **for** k = 0 **to** K $-$ 1 **do**
12: sum $\leftarrow$ sum + W_{in}[input][k] $\times$ W_{out}[target][k]
13: **end for**
14: grad $\leftarrow$ (label $-$ $sigmoid$(sum)) $\times$ α
15: **#pragma simd**
16: **for** k = 0 **to** K $-$ 1 **do**
17: temp $\leftarrow$ grad $\times$ W_{out}[target][k]
18: W_{out}[target][k] $\leftarrow$ W_{out}[target][k]+grad$\times W_{in}$[input][k]
19: W_{in}[input][k] $\leftarrow$ W_{in}[input][k] + temp
20: **end for**
21: **end if**
22: **end for**
23: **end for**

Algorithm 4 Repulsion() phase on **PAR-Word2Vec-cpu**

1: **// Update W_{out} and W_{in} with only shared negative samples**
2: label $\leftarrow$ 0, num_batch $\leftarrow$ L / B
3: **for** batch_id = 0 **to** num_batch $-$ 1 **do**
4: min_pos $\leftarrow$ batch_id $\times$ B, max_pos $\leftarrow$ min_pos + B $-$ 1
5: **if** (min_pos <= $C-1$) || ($L-1-$ min_pos <= $C-1$) **then**
6: $C_{rep} \leftarrow random\_uniform()$ % C
7: **else**
8: $C_{rep} \leftarrow random\_uniform()$ % (2 $\times$ C)
9: **end if**
10: $H \leftarrow T \times C_{rep}$
11: $shared\_ns$[0:$H-1$] $\leftarrow random\_uniform()$ % V
12: **memcpy**(M_{in}[0:$B-1$][0:$K-1$], W_{in}[$corpus$[sid][min_pos:min_pos+$B-1$]][0:$K-1$])
13: **memcpy**(M_{out}[0:$H-1$][0:$K-1$], W_{out}[$shared\_ns$[0:$H-1$]][0:$K-1$])
14: **// Multiplication of M_{out} and M_{in}^{T}**
15: M_{grad}[0:$H-1$][0:$B-1$]$\leftarrow$**sgemm**(M_{out}[0:$H-1$][0:$K-1$], M_{in}^{T}[0:$B-1$][0:$K-1$])
16: **// Compute gradient**
17: M_{grad}[0:$H-1$][0:$B-1$] $\leftarrow$ (label$-sigmoid(M_{grad}$[0:$H-1$][0:$B-1$]))$\times\alpha$
18: **// Multiplication of M_{grad} and M_{in}**
19: M_{out_up}[0:$H-1$][0:$K-1$]$\leftarrow$**sgemm**(M_{grad}[0:$H-1$][0:$B-1$], M_{in}[0:$B-1$][0:$K-1$])
20: **// Multiplication of M_{grad}^{T} and M_{out}**
21: M_{in_up}[0:$B-1$][0:$K-1$]$\leftarrow$**sgemm**(M_{grad}^{T}[0:$H-1$][0:$B-1$], M_{out}[0:$H-1$][0:$K-1$])
22: **Add**(W_{in}[$corpus$[sid][min_pos:min_pos+$B-1$]][0:$K-1$], M_{in_up}[0:$B-1$][0:$K-1$])
23: **Add**(W_{out}[$shared\_ns$[0:$H-1$]][0:$K-1$], M_{out_up}[0:$H-1$][0:$K-1$])
24: **end for**

A. Details of Parallel CPU implementation

The pseudo-codes for our parallel CPU implementation are shown in Algorithm 2, 3 and 4. The sets of sentences in the corpus are disjointly distributed for processing by different threads (line 4 in Algorithm 2). For each sentence, a *Repulsion* phase starts to process after the completion of all *Attraction* updates. As shown in Algorithm 3, a decoupled *Attraction* phase is performed in the same way as the original Word2Vec algorithm. Algorithm 4 shows the pseudo-code for the decoupled *Repulsion* phase. In the *Repulsion* phase, the number of shared negative samples H for the current mini-batch is determined by the first word's position in the mini-batch (lines 4-10). If the first word of the mini-batch is located at the start or end of the sentence, the corresponding mini-batch requires fewer shared negative samples, since the context window sizes of given words are small compared to the other words in the middle of the sentence. The H shared negative targets are then randomly selected in line 11. Next, the temporary matrices M_{in} and M_{out} for keeping input word vectors and shared negative target vectors are formed by pulling corresponding weight vectors from W_{in} and W_{out}, respectively (lines 12 and 13). Then all three matrix-matrix multiplications required for current mini-batch are efficiently performed using the **cblas_dgemm**() BLAS-3 routine provided in Intel's Math Kernel Library (MKL). In line 15 in Algorithm 4, the first matrix multiplication of two matrices, M_{out} and M_{in}^{T}, is performed to compute the gradients for the current mini-batch. The outputs of $M_{out} \cdot M_{in}^{T}$ are stored in an additional matrix M_{grad}. The update values for W_{out} are then computed by performing the second matrix multiplication of matrices M_{grad} and M_{in} (line 19) and storing the results in M_{out_up} matrix. Similarly, the update values for W_{in} can be obtained by performing the third matrix multiplication, with matrices M_{grad}^{T} and M_{out} (line 21). After completing all computations from the current mini-batch, the corresponding update values contained in M_{out_up} and M_{in_up} are accumulated to the main data structures, W_{out} and W_{in} (lines 22 and 23).

B. Details of Parallel GPU implementation

Algorithm 5 shows the pseudo-code for GPU implementation on the host for launching a GPU kernel (lines 6-16). In order to achieve massive parallelism on GPUs, we divide the corpus of sentences into different thread blocks (line 4 in Algorithm 5). Hence, the thread blocks are processed in parallel, whereas multiple sentences are processed sequentially within each thread block (lines 9-16). In line 2, we launch $\gamma \times 56$ thread blocks as there are 56 Streaming Multiprocessors (SMs) on an NVIDIA Pascal P100 GPU. γ is the overbooking factor used to maintain good load balance. Note that the parameter γ is varied according to the total number of sentences over the corpus. In order to simultaneously update K dimensions

of each word vector involved in the *Attraction* and *Repulsion* phases, K threads are selected within a thread block (line 3).

Algorithm 5 Parallel GPU implementation (**PAR-Word2Vec-gpu**)

```
1: λ ← α / total number of epochs
2: num_blocks ← γ × 56
3: num_threads ← K
4: num_sen_per_block ← (S+num_blocks−1) / num_blocks
5: repeat
6:     __shared__ shared_vector[1024]
7:     start_sen_id_per_block ← blockIdx.x × num_sen_per_block
8:     end_sen_id_per_block←start_sen_id_per_block+num_sen_per_block
9:     for sid = start_sen_id_per_block to end_sen_id_per_block do
10:        start_idx ← sen_ptr[sid]
11:        end_idx ← sen_ptr[sid+1]
12:        L ← end_idx − start_idx // L: length of sentence
13:        Attraction()
14:        __syncthreads()
15:        Repulsion()
16:    end for
17:    α ← α − λ
18: until convergence
```

Algorithm 6 Repulsion() phase on **PAR-Word2Vec-gpu**

```
1: label ← 0, num_batch ← L / B
2: for batch_id = 0 to num_batch − 1 do
3:     min_pos ← start_idx + (batch_id × B), max_pos ← min_pos+B−1
4:     shared_C_rand[blockIdx.x] ← curand_uniform() × (2 × C)
5:     H ← shared_C_rand[blockIdx.x] × T
6:     shared_ns[blockIdx.x×H:blockIdx.x×H+H−1] ← curand()×V
7:     memcpy(M_in[blockIdx.x×B×K+
       (((min_pos:max_pos−1)−min_pos)×K)+threadIdx.x],
       W_in[corpus[(min_pos:max_pos−1)]×K+threadIdx.x])
8:     memcpy(M_out[blockIdx.x×H×K+(0:H−1)×K+threadIdx.x],
       W_out[shared_ns[blockIdx.x×H+(0:H−1)]×K+threadIdx.x])
9:     // Efficient 2D-tiled matrix multiplication of M_out and M_in^T
10:    M_grad ← MatrixMultiplication(M_out, M_in^T)
11:    __syncthreads()
12:    M_grad[blockIdx.x×H×B:blockIdx.x×H×B+H−1] ←
       (label −
       sigmoid(M_grad[blockIdx.x×H×B:blockIdx.x×H×B+H−1]))×α
13:    __syncthreads()
14:    // Efficient 2D-tiled matrix multiplication of M_grad and M_in
15:    M_out_up ← MatrixMultiplication(M_grad, M_in)
16:    __syncthreads()
17:    // Efficient 2D-tiled matrix multiplication of M_grad^T and M_out
18:    M_in_up ← MatrixMultiplication(M_grad^T, M_out)
19:    __syncthreads()
20:    atomicAdd(W_in[corpus[(min_pos:max_pos−1)]×K+threadIdx.x],
       M_in_up[blockIdx.x×B×K+(((min_pos:max_pos−1)−min_pos)×K)
       +threadIdx.x])
21:    atomicAdd(W_out[shared_ns[blockIdx.x×H+(0:H−1)]×K+
       threadIdx.x],
       M_out_up[blockIdx.x×H×K+(0:H−1)×K+threadIdx.x])
22: end for
```

In the *Attraction* phase, to obtain the gradient of vector-vector multiplications of W_{in} and W_{out}, a warp-level reduction is performed by warp shuffling primitives. All the threads in the warp read $32/K$ of the word vectors (K dimensions) from W_{in} and W_{out} and perform multiplications. Then the computed gradients are stored in a single space of shared-memory (shared_vector[0]) since all the threads in each thread block must use the same gradient before computing the update values. The update values for each word vector are then accumulated to W_{in} and W_{out} in global memory using atomic operations, because multiple thread blocks can update the same word vector at the same time. At the end of the *Attraction* phase, all threads are synchronized before the start of the *Repulsion* phase (line 14 in Algorithm 5). Algorithm 6 shows the pseudo-code for the *Repulsion* phase within a GPU kernel. Similar to the *Repulsion* phase in our parallel CPU implementation, the number of shared negative samples H are chosen by the first word's position in the mini-batch (lines 3-5). After randomly selecting shared negative samples (line 6) and forming the input and output matrices by copying the corresponding word vectors from W_{in} and W_{out} for the current batch (lines 7 and 8), three matrix-matrix multiplications are performed using a 2D register tiling strategy along with the use of shared-memory (lines 10, 15 and 18). To achieve good performance on GPUs, the judicious use of shared-memory is crucial. Due to the limited amount of shared-memory per SM, we use 1024 size of shared-memory, which provides the best performing occupancy in the current NVIDIA Pascal GPU (line 6 in Algorithm 5). In order take advantage of warp execution, the two sub-matrices involved in each 2D-tiled multiplication are carefully partitioned into allocated shared memory space. For example, M_{out} and M_{in} matrices involved in the first matrix multiplication are divided into $(H \times K)$ / (16×32) and $(B \times K)$ / (16×32) tiles, respectively. The shared-memory and register tile sizes have an impact on both data reuse and concurrency. The higher the tile sizes the higher the data reuse. However, higher tile sizes demand more resources and thus limit the number of concurrently active threads (occupancy). The tile sizes were chosen such that the data movement was minimized while maintaining good concurrency.

C. Data Movement Analysis for PAR-Word2Vec

$$S(L(Attractions) + \frac{L}{B}(Repulsions)) =$$

$$S(L(1+2(C-C_{rand})(1+8K)) + \frac{L}{B}(\frac{6HBK}{\sqrt{\tau}} + H + 4BK + 4HK + 2HB)) \quad (6)$$

$$S(\frac{L}{B}(\frac{6HBK}{\sqrt{\tau}} + H + 4BK + 4HK + 2HB)) \quad (7)$$

We analyzed the data movement of our PAR-Word2Vec algorithm based on Algorithms 2, 3 and 4. Our algorithm iterates over all S sentences, and *Attraction* and *Repulsion* phases are separately performed within each sentence (lines 10 and 11 in Algorithm 2). The data movement cost of the decoupled *Attraction* phase is similar to the original SGNS algorithm without a negative sampling loop. The loop in line 3 of Algorithm 3 iterates over all L word tokens in each sentence and has an associated data movement of $1 + (2C - 2C_{rand})(1 + 2K + 4K + 2K)$ for the *Attraction* phase. In the *Repulsion* phase, the loop in line 3 in Algorithm 4 iterates over $\frac{L}{B}$ mini-batches for the current sentence. First, the H shared negative samples are randomly chosen and kept in the $shared\_ns$ array through the loop in line 11 (H writes). Then the shared negative word vectors and input word vectors pulled from W_{out} and W_{in} matrices are copied to M_{out} and M_{in} temporary matrices in lines 12 and 13 ($KB + KH$ reads and $KB + KH$ writes). Given M_{out} and M_{in} matrices, three matrix multiplications are required to perform *Repulsion* updates. It is well known that the highest order term in the number of data elements moved (between main memory and a cache of size τ words) for efficient tiled matrix multiplication

of two matrices A, $(M \times K)$ and B, $(K \times N)$ is $2MNK/\sqrt{\tau}$ (An extensive discussion of both lower bounds and data movement volume for several tiling schemes may be found in the recent work of Smith [25]). Hence, the data movement costs associated with the three matrix multiplications in lines 15, 19 and 21 are $2HKB/\sqrt{\tau}$, $2HBK/\sqrt{\tau}$ and $2BHK/\sqrt{\tau}$, respectively. The loop in line 17 computes the gradients and has an associated data movement cost of HB reads and HB writes. At the end of the *Repulsion* phase (lines 22 and 23), the update vectors in M_{out} and M_{in} matrices are accumulated to W_{out} and W_{in} ($KB + KH$ reads and $KB + KH$ writes). In total, Equation 6 shows the data movement cost for our PAR-Word2Vec algorithm, where τ is the cache size. Also, the data movement required for only *Repulsion* phase of PAR-Word2Vec is shown in Equation 7.

D. Data Movement Analysis Comparison

For the One Billion Word Benchmark dataset (S = 30,607,741 and $L = N/S$ = 804,269,958 / 30,607,741) with $K = 128$, $C = 8$, $T = 5$, $C_{rand} = 0$, $C_{rep} = 16$, $H = TC_{rep} = 80$, and $B = 24$ on a machine with 35 MB cache[2], based on Equation 4, the total data movement cost of original SG-NS based Word2Vec algorithm is 85,665,204$\times 10^6$ bytes. Whereas, based on Equation 6, the total data movement cost for our PAR-Word2Vec algorithm is only 15,109,321$\times 10^6$ bytes which is approximately 5.67$\times$ lower than the original SG-NS algorithm. Moreover, based on the Equation 5 and 7, the data movement associated with only *Repulsion* updates is greatly reduced by $1 - 19{,}239{,}278 \times 10^5$ / $72{,}487{,}241 \times 10^6$ $\approx$ 97.3%. On the other hand, both algorithms must have the same amount of data movements for *Attraction* updates. The data movement improvement of our approach can be clearly proven by the fact that the difference between Equation 4 and 5 (*Attraction* updates for original SG-NS based Word2Vec), and the difference between Equation 6 and 7 (*Attraction* updates for PAR-Word2Vec) are exactly matched.

VI. Experimental Evaluation

This section provides both performance and quality assessments for the Word2Vec and Node2Vec algorithms. Our PAR-Word2Vec implementations on multi-core CPUs and GPUs are compared with various state-of-the-art implementations.

A. Benchmarking Machines

The detailed configuration of the benchmarking machines used for experiments is shown in Table II. All the CPU experiments were run on an Intel Xeon CPU E5-2680 v4 running at 2.4 GHz with 128 GB RAM. The GPU experiments were run on an NVIDIA Tesla P100 PCIE GPU with 16 GB global memory.

B. Datasets

Table III and IV show the characteristics of each text and graph dataset and the details of graph datasets, respectively. For the direct Word2Vec evaluations, we used two publicly available real-world text datasets – text8[3] and One Billion Word Benchmark (1B-Word)[4]. text8 contains approximately 17 million word tokens from Wikipedia. One Billion Word Benchmark (1B-Word) corpus contains approximately 0.8 billion word tokens from the WMT 2011 News Crawl data. In order to evaluate the Node2Vec algorithm, we also used five publicly available graph datasets: three labeled datasets – BlogCatalog[5], PPI[6] and Wikipedia-2006[7] – and two unlabeled datasets – Facebook[8] and arXiv ASTRO-PH (ASTRO-PH)[9]. BlogCatalog is derived from friend and group information on the BlogCatalog website. Protein-Protein Interactions (PPI) dataset is a subgraph of the PPI network for Homo Sapiens. Wikipedia-2006 dataset is derived from word cooccurrence information from the first 10^9 bytes of English Wikipedia on March 3, 2006. The Facebook graph encodes friend relations between Facebook users. arXiv ASTRO-PH (ASTRO-PH) dataset is a collaboration network of Astro Physics related papers in the e-print arXiv between January 1993 to April 2003. Given the directed/undirected graph which represents the connections between nodes with/without weights, random walks are pre-generated with the original Node2Vec implementation released by SNAP (Stanford Network Analysis Platform)[10]. Thereafter, the pre-generated random walks are used as inputs for all the Word2Vec variants. Each random walk is comprised of a sequence of $N/S = 81$ nodes.

TABLE II: Machine configuration

Machine	Details
CPU	Intel(R) Xeon(R) CPU E5-2680 v4; 14 cores and 28 threads, 128 GB RAM, 76.8 GB/s bandwidth, 35 MB L2 cache; ICC 18.0.3
GPU	Tesla P100 PCIE; 56 SMs, 64 cores/SM, 16 GB Global Memory, 732 GB/s bandwidth, 4 MB L2 cache; CUDA 9.2.88

TABLE III: Statistics of text datasets, and graph datasets pre-generated by Node2Vec. V is the number of unique words/nodes, S is the total number of sentences over the corpus/the total number of random walks over the graph, and N is the total number of word tokens over the corpus/the sum of the length of the walks in S.

Dataset		*V*	*S*	*N*
Text Dataset	text8	71,291	9,385	16,718,843
	1B-Word	555,514	30,607,741	804,269,958
Graph Dataset	BlogCatalog	10,313	103,120	8,352,720
	PPI	3,891	38,900	3,150,900
	Wikipedia-2006	4,778	47,770	3,869,370
	Facebook	4,040	40,390	3,271,590
	ASTRO-PH	18,773	187,720	15,205,320

C. Evaluation Metrics

1) Word2Vec Evaluation Metrics: We used standard word similarity and relatedness evaluations to compare word embeddings learned from PAR-Word2Vec and other methods [26].

[2] $C_{rand} = 0$ makes the maximum size of context window at current center word according to the lines 8-9 in Algorithm 1. $C_{rep} = 16$ corresponds to the maximum number of C_{rep}, when $C = 8$ according to the line 8 in Algorithm 4. $B = 24$ is chosen to be used for all the experiments in Section VI.

[3] http://mattmahoney.net/dc/text8.zip

[4] http://www.statmt.org/lm-benchmark/

[5] http://socialcomputing.asu.edu/datasets/BlogCatalog3

[6] https://downloads.thebiogrid.org/BioGRID

[7] http://mattmahoney.net/dc/textdata.html

[8] http://snap.stanford.edu/data/egonets-Facebook.html

[9] http://snap.stanford.edu/data/ca-AstroPh.html

[10] https://github.com/snap-stanford/snap/tree/master/examples/node2vec

TABLE IV: Details of graph datasets. E is the total number of edges and A is the number of different labels for nodes in the graph.

Dataset	Graph type	*E*	*A*
BlogCatalog	undirected/unweighted	333,983	39
PPI	undirected/unweighted	76,584	50
Wikipedia-2006	undirected/weighted	184,812	40
Facebook	undirected/unweighted	88,234	N/A
ASTRO-PH	directed/unweighted	198,110	N/A

This task involves inventories of word pairs that have been assigned a similarity or relatedness score by human annotators (e.g., (tiger, cat, 7.35), (stock, life, 0.92) [27]). For each word pair, the cosine similarity of the corresponding embeddings is calculated. These cosine similarities are then rank-ordered, and the rankings compared to rank-ordering of the human judgments using Pearson's ρ (-1 to 1, higher is better). We evaluated on the following datasets:

– **WordSim-353:** 353 pairs rated for similarity of meaning [28].
– **SimLex-999:** 999 pairs rated specifically for similarity, and not relatedness [28].

These evaluations are referred to as "intrinsic", since they do not use any learned parameters beyond the word embeddings. Analogy completion tasks [1] have often been used as another "intrinsic" evaluation, together with similarity/relatedness. However, these tasks have well-documented issues that limit their value as an evaluation metric [29], [30]. We therefore followed prior work [31], [32] by augmenting our intrinsic evaluation with "extrinsic" evaluations that measure the quality of word embeddings by plugging them into another machine learning model that learns to use them as features. We evaluated on the following tasks:

– **Relation extraction:** SemEval-2010 shared task 8 [33], using a CNN model with word and distance embeddings [34] for nine-way relation classification.
– **Sentiment analysis:** positive/negative binary classification of IMDB movie reviews [35], using a single 100-dimensional LSTM.

2) Node2Vec Evaluation Metrics: The quality of the trained node embeddings out of Word2Vec can be evaluated through such applications as multi-label classification (e.g., classifying bloggers into categories in BlogCatalog dataset) and link prediction tasks. [7], [8], [16]. The node embeddings coming out of the various Word2Vec implementations can just be fed into the same classification models to evaluate them.

– **Multi-label classification**: For the evaluations with labeled graph datasets, the logistic regression model takes the $V \times K$ node embeddings as input and learns the model based on a logistic function. The trained model predicts the probability of each class given test embedding vectors.
– **Link prediction**: We conducted link prediction task with unlabeled graph datasets based on the edge information. Link prediction can be considered as a binary classification by predicting the connection between two nodes. Where two nodes are connected in the graph, they were labeled as a positive example. Negative examples were generated by randomly sampling pairs of nodes not connected to each other. For a fair evaluation, the numbers of positive and negative examples used in our experiments were balanced and we used the same negative samples across all variants of Word2Vec. Features were generated for each sample by concatenating the learned embeddings for each pair of nodes; these features were then plugged into Support-Vector Machine (SVM) model [36] for training and testing.

D. Word2Vec Implementations Compared

We evaluated PAR-Word2Vec on multi-core CPUs and GPUs with state-of-the art parallel Word2Vec implementations. The seven implementations used in our comparisons are as follows: Word2Vec-cpu [2], pWord2Vec-cpu [18], wombatSGNS-cpu [20], pSGNScc-cpu [21], accSGNS-gpu [23], and our PAR-Word2Vec-cpu and PAR-Word2Vec-gpu. Note that, all compared models are based on SG-NS algorithm. While Word2Vec-cpu uses Pthreads API, all other CPU implementations, including pWord2Vec-cpu, wombatSGNS-cpu, pSGNScc-cpu, and our PAR-Word2Vec-cpu, use OpenMP API and the same Intel's Math Kernel Library (MKL) for all BLAS (Basic Linear Algebra Subprograms) operations. It is obvious that annealing the learning rate is a critical part of having good quality of embeddings. For all CPU implementations, during training the model for each epoch, the corpus is read sentence-by-sentence and the learning rate is reduced based on this reading progress through the corpus. Whereas, our PAR-Word2Vec-gpu completes the reading of the entire corpus and copies it into GPU memory once prior to the start of training and gradually decreases the learning rate at the end of each epoch. As shown in line 17 in Algorithm 5, the learning rate is decreased by $\alpha = \alpha - \textit{initial } \alpha \ / \ \textit{total number of epochs}$ at the end of each epoch.

E. Performance Evaluation

For all datasets, including text and graph, we used the same number of negative samples $T = 5$ and same size of embedding vector $K = 128$. For text and graph datasets, respectively, we set the window size C to 8 and 10. We then trained all variants of Word2Vec over 10 epochs with all datasets. To ensure fairness, 28 threads were used in all CPU experiments. In addition, all the parameters in all CPU and GPU implementations were tuned for each dataset and the best performing configurations were selected. For both PAR-Word2Vec-cpu and PAR-Word2Vec-gpu, we used $B = 24$ for all the experiments. However, the number of thread blocks were varied according to the total number of sentences in each dataset; $\gamma = 64$ for text8, BlogCatalog, PPI, Wikipedia-2006, Facebook and ASTRO-PH, and $\gamma = 1024$ for 1B-Word datasets. As suggested in the running scripts of pWord2Vec-cpu and pSGNScc-cpu, the batch size of both pWord2Vec-cpu and pSGNScc-cpu were set to $2 \times C + 1$. In the process of pre-generating graph datasets, we used the same p and q values for BlogCatalog, PPI and Wikipedia datasets as suggested in [7]. Note that the window size C is randomly selected for each inner loop in Word2Vec algorithm (e.g., line 8 in Algorithm 1). Furthermore, all negative target words are randomly chosen from the vocabulary (e.g., line 17 in Algorithm 1). Therefore, all the experiment results presented in this section are averaged over 5 different executions.

1) Intrinsic Evaluations of Word Embeddings: Figure 5 shows word similarity scores over training epochs, comparing all variants of Word2Vec implementations on text datasets. As shown in Table V, the difference between our PAR-Word2Vec models and the baselines is not statistically significant in terms

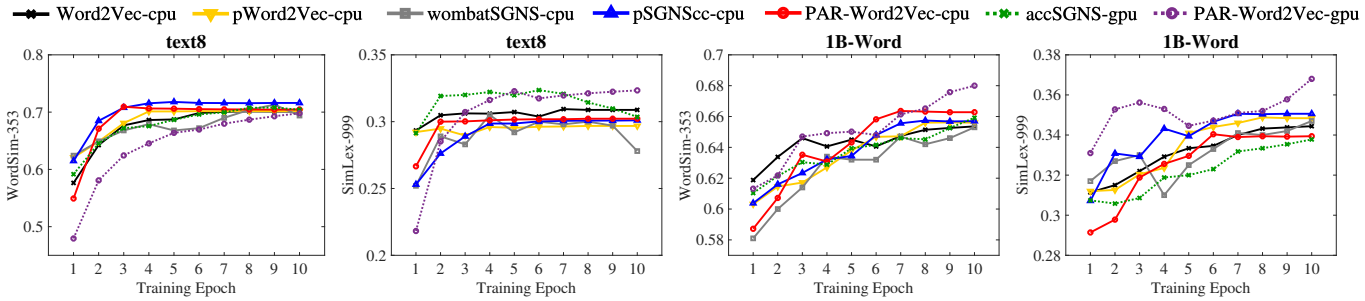


Fig. 5: Comparison of word similarity scores over training epoch on text datasets, $K = 128$. Each point is averaged over five executions. X-axis: number of training epochs; Y-axis: word similarity scores.

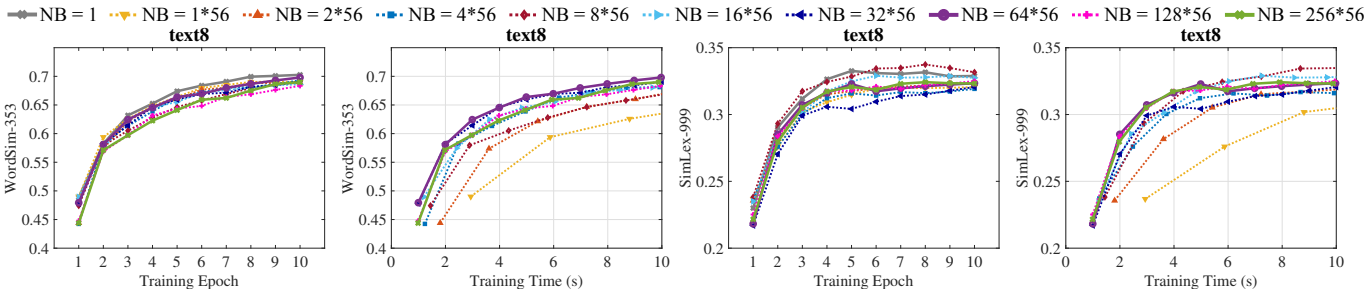


Fig. 6: Word similarity scores over training epoch and time across different number of thread blocks used in PAR-Word2Vec-gpu on text8 dataset, $K = 128$. Each point is averaged over five executions. NB: the total number of thread blocks; X-axis: number of training epochs and training time in seconds; Y-axis: word similarity scores.

of the converged word similarity scores after training for 10 epochs. For the large 1B-Word dataset, our PAR-Word2Vec-cpu and PAR-Word2Vec-gpu consistently produced high word similarity scores on WordSim-353 and SimLex-999 tasks. This result demonstrate that our sentence-wise decoupled Attraction-Repulsion based approach is highly beneficial to performance improvement in terms of both model quality and speedup. In the large 1B-Word dataset, 27,994,959 / 30,607,741 $\approx$ 91.46 % of sentences over the corpus include less than 25 words. The mini-batch size $B = 24$ that we used for the experiment with 1B-Word dataset indicates that sharing the same negative samples for all words within a sentence would not affect the quality of model at all.

TABLE V: Mean and standard deviation of converged WordSim-353 and SimLex-999 scores over 5 different executions on text datasets.

Model	text8		1B-Word	
	WordSim-353	SimLex-999	WordSim-353	SimLex-999
Word2Vec-cpu	0.701 (±0.010)	0.308 (±0.014)	0.653 (±0.004)	0.344 (±0.007)
pWord2Vec-cpu	0.701 (±0.008)	0.296 (±0.008)	0.656 (±0.003)	0.348 (±0.002)
wombatSGNS-cpu	0.694 (±0.013)	0.278 (±0.009)	0.653 (±0.001)	0.350 (±0.002)
pSGNScc-cpu	0.716 (±0.008)	0.301 (±0.009)	0.657 (±0.003)	0.350 (±0.001)
PAR-Word2Vec-cpu	0.705 (±0.008)	0.302 (±0.003)	0.663 (±0.002)	0.341 (±0.003)
accSGNS-gpu	0.704 (±0.004)	0.303 (±0.002)	0.659 (±0.003)	0.337 (±0.002)
PAR-Word2Vec-gpu	0.698 (±0.008)	0.323 (±0.005)	0.680 (±0.004)	0.368 (±0.004)

On the small text8 dataset, PAR-Word2Vec-gpu has some variability; it produces high averaged score for the SimLex-999 similarity task, but low averaged score for the WordSim-353 task as shown in Table V. However, PAR-Word2Vec-gpu has standard deviations that overlap in the ranges (e.g., $0.698 + 0.008 = 0.706$ on WordSim-353 task), implying that PAR-Word2Vec-gpu yields comparable results on all intrinsic evaluations. We suspect that issue is related to the number of thread blocks. As seen in Algorithm 5, each thread block is processing multiple sentences and the global W_{in} and W_{out} matrices are updated after each sentence is processed (local updates within a sentence; global updates across sentences). Therefore, different thread blocks are not able to see each others update until the entire sentences are processed as shown in Algorithm 6. Hence, the updates would not be the same as sequential. The more the number of thread blocks the higher the chance of this incoherence. Especially for the small text8 dataset, this is an issue as average sentence length is around 1000 words (higher the sentence length higher the chance of this incoherence). For the large 1B-Word dataset, the average sentence length is less than 25 and therefore this effect may not be visible for 1B-Word dataset. In order to verify this issue, we conducted an experiment with varying the number of thread blocks in text8 dataset. As shown in Figure 6, the results were mostly matched as we expected: the smaller number of thread blocks tends to provide a better quality, but slower training time. Another possible reason is that the method of annealing learning rate in PAR-Word2Vec-gpu is different from all other variants (see Section VI.D). Accordingly, to achieve high performance in terms of both quality and training time, we had chosen to use 64×56 thread blocks and 1024×56 thread blocks for text8 and 1B-Word datasets, respectively.

2) Extrinsic Evaluations of Word Embeddings: Table VI shows performance on the extrinsic relation extraction and sentiment classification tasks after training for 10 epochs; performance numbers were averaged over five replicates each of five embedding runs, to control for random initialization

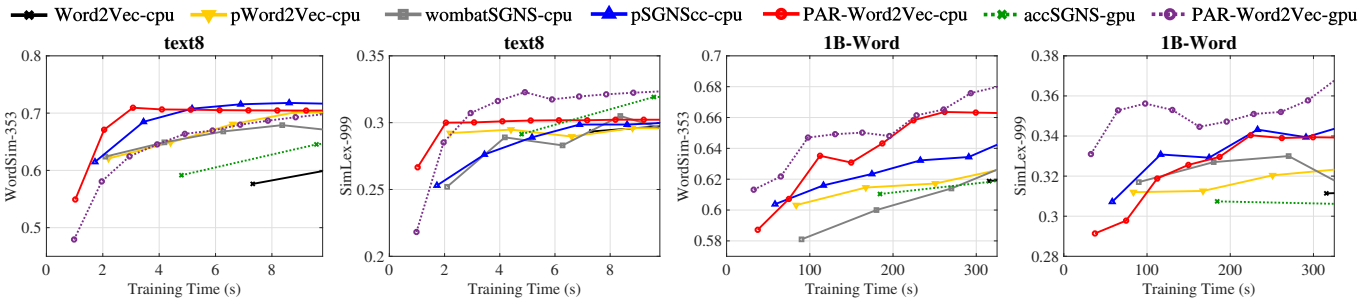


Fig. 7: Comparison of word similarity scores over training time on text datasets, $K = 128$. Each point is averaged over five executions. X-axis: training time in seconds; Y-axis: word similarity scores.

TABLE VI: Mean and standard deviation of Macro F_1 scores for relation extraction task and accuracy for sentiment analysis task over 5 different runs each for 5 different embedding executions, by Word2Vec variants.

Model	text8		1B-Word	
	Relation Extraction	Sentiment Analysis	Relation Extraction	Sentiment Analysis
Word2Vec-cpu	0.671 (±0.010)	0.795 (±0.006)	0.689 (±0.009)	0.782 (±0.008)
pWord2Vec-cpu	0.669 (±0.006)	0.791 (±0.004)	0.686 (±0.008)	0.779 (±0.007)
wombatSGNS-cpu	0.666 (±0.007)	0.776 (±0.005)	0.691 (±0.010)	0.783 (±0.005)
pSGNScc-cpu	0.666 (±0.009)	0.790 (±0.005)	0.685 (±0.010)	0.784 (±0.006)
PAR-Word2Vec-cpu	0.665 (±0.010)	0.783 (±0.008)	0.691 (±0.008)	0.780 (±0.004)
accSGNS-gpu	0.680 (±0.010)	0.796 (±0.007)	0.689 (±0.006)	0.787 (±0.006)
PAR-Word2Vec-gpu	0.663 (±0.010)	0.807 (±0.004)	0.623 (±0.009)	0.780 (±0.004)

effects in both embedding learning and the models used for extrinsic evaluations. PAR-Word2Vec-cpu matches the evaluation quality of other CPU implementations, using both text8 and 1B-word datasets. PAR-Word2Vec-gpu yields comparable or superior performance on sentiment classification, but relation extraction quality drops by 6% with the larger 1B-word dataset; however, quality with text8 is on par with other implementations. Taken together with the intrinsic evaluation results, this suggests that the massive parallelization of our PAR-Word2Vec-gpu algorithm may be exploring the limits of HOGWILD!-style data parallelization in word embedding training.

TABLE VII: Mean and standard deviation of Micro F_1 and Macro F_1 scores for multi-label classification, and Micro F_1 score for link prediction task over 5 different runs each for 5 different embedding executions, by Word2Vec variants.

Model	BlogCatalog		PPI		Wikipedia-2006		Facebook	ASTRO-PH
	Micro F_1	Macro F_1	Micro F_1	Macro F_1	Micro F_1	Macro F_1	Micro F_1	Micro F_1
Word2Vec-cpu	0.292	0.192	0.120	0.094	0.338	0.071	0.699	0.723
pWord2Vec-cpu	0.290	0.188	0.124	0.095	0.338	0.070	0.691	0.721
wombatSGNS-cpu	0.291	0.189	0.102	0.079	0.336	0.069	0.692	0.718
pSGNScc-cpu	0.290	0.189	0.120	0.093	0.323	0.054	0.686	0.692
PAR-Word2Vec-cpu	0.290	0.190	0.124	0.096	0.340	0.074	0.687	0.723
accSGNS-gpu	0.279	0.188	0.119	0.093	0.334	0.072	0.698	0.721
PAR-Word2Vec-gpu	0.280	0.186	0.121	0.092	0.326	0.070	0.672	0.720
Avg. standard dev.	±0.001	±0.001	±0.007	±0.006	±0.002	±0.002	±0.001	±0.002

3) Extrinsic Evaluations of Graph Embeddings: To evaluate the multi-label classification task with labeled graph datasets, we measured Micro F_1 and Macro F_1 scores through 10-fold cross-validation. For the link prediction task with unlabeled graph datasets, we only measured Micro F_1 score since the numbers of distinct positive edges and distinct negative edges used for the SVM training were the identical. As shown in Table VII, all variants including our CPU and GPU implementations maintain mostly the same quality of node embeddings.

4) Speedup: Figure 7 shows the word similarity scores over elapsed training time with text datasets. Our PAR-Word2Vec-cpu and PAR-Word2Vec-gpu achieved significant improvement in performance over existing state-of-the-art parallel Word2Vec implementations while maintaining the same evaluation quality. As the results in Table VIII show, PAR-Word2Vec-cpu achieved approximately 9.01×, 2.41×, 2.51×, and 1.62× speedup on the large 1B-Word dataset compared to Word2Vec-cpu, pWord2Vec-cpu, wombatSGNS-cpu, and pSGNScc-cpu, respectively. For GPU implementations, our parallel PAR-Word2Vec-gpu launches a kernel with only $\gamma \times 56$ thread blocks with K threads and accSGNS-gpu launches a large amount of S thread blocks along with K threads. Although it uses a relatively fewer thread blocks compared to accSGNS-gpu, PAR-Word2Vec-gpu achieved 5.96× speedup over accSGNS-gpu while maintaining same or better quality. With the graph datasets, the range of performance improvements of our CPU and GPU implementations is almost the same as the performance improvement with text datasets. On the large ASTRO-PH graph dataset, PAR-Word2Vec-cpu achieved 8.77×, 1.86×, 2.01×, and 1.93× speedup compared to Word2Vec-cpu, pWord2Vec-cpu, wombatSGNS-cpu, and pSGNScc-cpu, respectively. Our PAR-Word2Vec-gpu also consistently outperformed accSGNS-gpu. One of the major factors that limits our single GPU implementation compared to our single CPU implementation is synchronization overheads. Our GPU implementation uses shared-memory to keep a slice of W_{in}, W_{out} and temporary results. This requires multiple synchronizations (one synchronization per load, store and update of each data structure). In contrast, our CPU implementation does not require any synchronization as we are using the implicit cache to buffer data. Another factor that affects the GPU performance is atomic operations. Since the amount of parallelism in GPUs is much higher than CPUs, we had to use atomic updates to maintain consistency of our data structures and this negatively impacted performance. We also found that intra thread-block load imbalance also limited the GPU performance. Techniques like binning can be employed

TABLE VIII: Comparison of the training time in seconds per epoch on text and graph datasets.

Model	Text Dataset		Labeled Graph Dataset			Unlabeled Graph Dataset	
	text8	1B-Word	BlogCatalog	PPI	Wikipedia-2006	Facebook	ASTRO-PH
Word2Vec-cpu	7.32	315.46	6.43	3.13	2.95	2.55	12.54
pWord2Vec-cpu	2.20	86.63	1.56	0.45	0.55	0.53	2.66
wombatSGNS-cpu	2.09	90.04	1.43	0.47	0.58	0.71	2.88
pSGNScc-cpu	1.72	58.20	1.46	0.70	0.75	0.84	2.77
PAR-Word2Vec-cpu	**1.02**	**37.43**	**0.83**	**0.33**	**0.28**	**0.31**	**1.43**
accSGNS-gpu	4.79	185.31	2.23	0.66	0.62	1.37	6.44
PAR-Word2Vec-gpu	**0.98**	**32.60**	**0.72**	**0.20**	**0.21**	**0.27**	**1.08**

to reduce load imbalance.

VII. Conclusion

In this paper, we built a parallel word embedding algorithm to enhance data locality, focusing on reduction of data movement. To achieve high performance, minimizing data movement is a critical factor, since data movement is much more expensive than arithmetic operations. We found the main bottleneck of the original Word2Vec algorithm by conducting a systematic analysis of data movement. The rearrangement of data computation enables our proposed algorithm to greatly reduce the data movement overheads. Experiments on the large datasets show that our algorithm achieves superior performance over the existing state-of-the-art implementations. We also presented insights into parallelism versus data-locality trade-offs as well as performance versus quality trends.

References

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[2] 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*, 2013, pp. 3111–3119.

[3] D. Zeng, K. Liu, Y. Chen, and J. Zhao, "Distant supervision for relation extraction via piecewise convolutional neural networks," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1753–1762.

[4] P. Nakov, A. Ritter, S. Rosenthal, F. Sebastiani, and V. Stoyanov, "Semeval-2016 task 4: Sentiment analysis in twitter," in *Proceedings of the 10th international workshop on semantic evaluation (semeval-2016)*, 2016, pp. 1–18.

[5] S. Moen and T. S. S. Ananiadou, "Distributional semantics resources for biomedical text processing," *Proceedings of LBM*, pp. 39–44, 2013.

[6] M. Habibi, L. Weber, M. Neves, D. L. Wiegandt, and U. Leser, "Deep learning with word embeddings improves biomedical named entity recognition," *Bioinformatics*, vol. 33, no. 14, pp. i37–i48, 2017.

[7] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 855–864.

[8] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 701–710.

[9] D. Newman-Griffis and A. Zirikly, "Embedding Transfer for Low-Resource Medical Named Entity Recognition: A Case Study on Patient Mobility," in *Proceedings of the BioNLP 2018 workshop*. Melbourne, Australia: Association for Computational Linguistics, jul 2018, pp. 1–11.

[10] W. L. Hamilton, J. Leskovec, and D. Jurafsky, "Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, aug 2016, pp. 1489–1501.

[11] A. Kutuzov, E. Velldal, and L. Øvrelid, "Tracing armed conflicts with diachronic word embedding models," in *Proceedings of the Events and Stories in the News Workshop*. Vancouver, Canada: Association for Computational Linguistics, aug 2017, pp. 31–36.

[12] M. Sahlgren and A. Lenci, "The effects of data size and frequency range on distributional semantic models," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2016, pp. 975–980.

[13] O. Levy, Y. Goldberg, and I. Dagan, "Improving distributional similarity with lessons learned from word embeddings," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015.

[14] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2177–2185.

[15] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model." in *Aistats*, vol. 5. Citeseer, 2005, pp. 246–252.

[16] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.

[17] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[18] S. Ji, N. Satish, S. Li, and P. Dubey, "Parallelizing word2vec in shared and distributed memory," *arXiv preprint arXiv:1604.04661*, 2016.

[19] J. B. Vuurens, C. Eickhoff, and A. P. de Vries, "Efficient parallel learning of word2vec," *arXiv preprint arXiv:1606.07822*, 2016.

[20] T. M. Simonton and G. Alaghband, "Efficient and accurate word2vec implementations in gpu and shared-memory multicore architectures," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[21] V. Rengasamy, T.-Y. Fu, W.-C. Lee, and K. Madduri, "Optimizing word2vec performance on multicore systems," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2017, p. 3.

[22] E. Ordentlich, L. Yang, A. Feng, P. Cnudde, M. Grbovic, N. Djuric, V. Radosavljevic, and G. Owens, "Network-efficient distributed word2vec training system for large vocabularies," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1139–1148.

[23] S. Bae and Y. Yi, "Acceleration of word2vec using gpus," in *International Conference on Neural Information Processing*. Springer, 2016, pp. 269–279.

[24] J. Canny, H. Zhao, B. Jaros, Y. Chen, and J. Mao, "Machine learning at the limit," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 233–242.

[25] T. M. Smith *et al.*, "Theory and practice of classical matrix-matrix multiplication for hierarchical memory architectures," Ph.D. dissertation, 2018.

[26] T. Schnabel, I. Labutov, D. Mimno, and T. Joachims, "Evaluation methods for unsupervised word embeddings," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2015, pp. 298–307.

[27] H. Rubenstein and J. B. Goodenough, "Contextual correlates of synonymy," *Communications of the ACM*, vol. 8, no. 10, pp. 627–633, 1965.

[28] F. Hill, R. Reichart, and A. Korhonen, "Simlex-999: Evaluating semantic models with (genuine) similarity estimation," *Computational Linguistics*, vol. 41, no. 4, pp. 665–695, 2015.

[29] A. Rogers, A. Drozd, and B. Li, "The (too Many) Problems of Analogical Reasoning with Word Vectors," *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017)*, pp. 135–148, 2017.

[30] D. Newman-Griffis, A. M. Lai, and E. Fosler-Lussier, "Insights into Analogy Completion from the Biomedical Domain," in *BioNLP 2017*. Vancouver, Canada: Association for Computational Linguistics, aug 2017, pp. 19–28.

[31] B. Chiu, A. Korhonen, and S. Pyysalo, "Intrinsic Evaluation of Word Vectors Fails to Predict Extrinsic Performance," *Proceedings of the 1st Workshop on Evaluating Vector Space Representations for NLP*, pp. 1–6, 2016.

[32] B. Whitaker, D. Newman-Griffis, A. Haldar, H. Ferhatosmanoglu, and E. Fosler-Lussier, "Characterizing the impact of geometric properties of word embeddings on task performance," in *Proceedings of the Third Workshop on Evaluating Vector Space Representations for NLP (RepEval)*. Minneapolis, MN: Association for Computational Linguistics, 2019.

[33] I. Hendrickx, S. N. Kim, Z. Kozareva, P. Nakov, D. Ó Séaghdha, S. Padó, M. Pennacchiotti, L. Romano, and S. Szpakowicz, "SemEval-2010 Task 8: Multi-Way Classification of Semantic Relations between Pairs of Nominals," in *Proceedings of the 5th International Workshop on Semantic Evaluation*. Association for Computational Linguistics, 2010, pp. 33–38.

[34] D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao, "Relation Classification via Convolutional Deep Neural Network," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. Dublin City University and Association for Computational Linguistics, 2014, pp. 2335–2344.

[35] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning Word Vectors for Sentiment Analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2011, pp. 142–150.

[36] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.