Optimizing Word2Vec Performance on Multicore Systems

Vasudevan Rengasamy
Tao-Yang Fu
Wang-Chien Lee
Kamesh Madduri
The Pennsylvania State University
Department of Computer Science and Engineering
University Park, Pennsylvania 16802
[vxr162,txf225,wlee,madduri]@cse.psu.edu

ABSTRACT

The Skip-gram with negative sampling (SGNS) method of Word2Vec is an unsupervised approach to map words in a text corpus to low dimensional real vectors. The learned vectors capture semantic relationships between co-occurring words and can be used as inputs to many natural language processing and machine learning tasks. There are several high-performance implementations of the Word2Vec SGNS method. In this paper, we introduce a new optimization called *context combining* to further boost SGNS performance on multicore systems. For processing the One Billion Word benchmark dataset on a 16-core platform, we show that our approach is 3.53× faster than the original multithreaded Word2Vec implementation and 1.28× faster than a recent parallel Word2Vec implementation. We also show that our accuracy on benchmark queries is comparable to state-of-the-art implementations.

CCS CONCEPTS

•**Computing methodologies** → *Shared memory algorithms; Natural language processing; Unsupervised learning;*

KEYWORDS

word embeddings, Word2Vec, SGD, multicore

ACM Reference format:
Vasudevan Rengasamy, Tao-Yang Fu, Wang-Chien Lee, and Kamesh Madduri. 2017. Optimizing Word2Vec Performance on Multicore Systems. In *Proceedings of IA^3 '17: Seventh Workshop on Irregular Applications: Architectures and Algorithms, Denver, CO, USA, November 12–17, 2017 (IA^3 '17),* 9 pages.
DOI: 10.1145/3149704.3149768

1 INTRODUCTION

Word embedding techniques learn vector representations of words in a given textual dataset such that semantically and syntactically relevant words are close to each other in the vector space. The learned word vectors are effective and discriminative as inputs to many natural language processing and machine learning applications such as document classification [14], machine translation [28], and named entity recognition [15]. Distributional Semantic Models (DSMs) are approaches that use the count of co-occurrences of words to obtain word representations. For example, the Positive Pointwise Mutual Information (PMI) method [8] learns high dimensional sparse vector representations of words using the co-occurrence counts of words. Each element in a word's vector gives the strength of association of that word to another word in the vocabulary. Another DSM approach is Singular Value Decomposition, where the dimensionality of the PMI matrix is reduced to create dense vector representations for words.

Bengio et al. [4] and Collobert and Weston [9] present neural network-based language models for predicting the next word when given a sequence of words. In contrast to the count-based DSM approaches, the neural network models predict words that are syntactically and semantically similar. Mikolov et al.'s Word2Vec [19] and Pennington et al.'s GloVe [22] are two popular neural network-based models for word representation learning.

The Word2Vec Skip-gram with negative sampling (SGNS) algorithm is widely used for learning word vectors that are useful for predicting the surrounding words (i.e., context words) of each word (i.e., a target word) in a sentence. Levy, Goldberg, and Dagan [16] show that SGNS training is faster than other competing methods. Word2Vec has received considerable attention in the natural language processing community. For example, Kiros et al. [13] propose Skip-thought vectors that use the Skip-gram model at the sentence level instead of word level to predict surrounding sentences that share the same semantic information as the target sentence.

Although SGNS is faster than alternatives, it can still take considerable time for training datasets with billions of words. For example, our experiment indicates that on a 16-core platform, SGNS takes nearly one hour to process a benchmark dataset with 805 million words. Text corpora of several billion words are now commonplace, and so improving SGNS efficiency is important. Also, any optimization technique that improves SGNS's throughput can be used to accelerate other applications of the Skip-gram model such as Skip-thought vectors and BioVec [3]. Hence, in this paper, we focus on improving the throughput of the Word2Vec SGNS algorithm.

SGNS uses the Stochastic Gradient Descent (SGD) algorithm for model parameter optimization. The input text corpus is scanned word by word to generate word pairs. A word pair consists of either a target word and another word from its neighborhood (a *positive sample*), or the target word and a randomly chosen word (a *negative*

DOI: 10.1145/3149704.3149768

sample). The probability of whether these two words co-occur in a sentence is estimated, and the vectors corresponding to the two words are updated, based on the gradients of the objective function with respect to the two words.

The main throughput-limiting step in SGNS is the probability calculation. This involves several vector-vector operations (e.g., the inner product of vectors corresponding to two words). The inner product of two length-D vectors requires $3D$ memory references and $2D$ floating point operations. Hence, the arithmetic intensity, or the number of floating point operations per memory access, is $\frac{2}{3}$. Without reuse, vector operations on modern platforms tend to be memory-bound. Moreover, in a multithreaded setting, threads update word vectors asynchronously without locking or atomics (based on the HOGWILD! scheme [21]). This could lead to a *ping-ponging of cache lines* [12].

In this paper, we propose a new SGNS optimization called *context combining*: we aim to improve the throughput of Word2Vec by simultaneously processing multiple contexts and reusing positive and negative samples. Due to the reuse across contexts, this optimization has the effect of converting vector-vector inner products used in SGNS to efficient matrix-matrix multiplications, thereby improving floating point throughput. The data reuse also reduces the overhead due to random memory accesses and asynchronous model parameter updates. For processing the One Billion Word benchmark dataset on a 16-core platform, we achieve a 3.53× speedup in comparison to the original Word2Vec implementation, and a 1.28× speedup compared to pWord2Vec [12], another recent parallel SGNS implementation.

2 BACKGROUND

2.1 Word2Vec training process

Word2Vec includes two model architectures to learn word representations, Contextual Bag-Of-Words (CBOW) and SGNS. CBOW aims to predict the target word given the surrounding words (or the context), whereas SGNS aims to predict context words given the target word. Prior evaluations show that SGNS performs better than CBOW on semantic tests, while performing slightly worse on syntactic tests [19]. Further, SGNS is shown to have a higher accuracy on infrequent words [2]. For these reasons, SGNS is widely used in many applications and hence is the focus of our work.

2.1.1 Skip-gram Model. The Skip-gram model is a single layer neural network model with one hidden layer. The input layer is a vector of size V (where V is the vocabulary size) representing a 1-hot encoding of words. The low dimensional word representations are stored as input-hidden layer weight matrix M_{in}, in which each row is a D-dimensional vector representation of the corresponding word in the vocabulary. The 1-hot encoding performs the function of selecting the input word representation from M_{in}. The output layer computes the probability of a word y occurring in the same context as a target word x, by computing probabilities using x and K randomly-chosen *negative samples*:

$$\log\ P(y|x) \approx \log \mathrm{sig}(\mathbf{v_x}^T \mathbf{v_y}) + \sum_{i=1}^{K} \log \mathrm{sig}(-\mathbf{v_x}^T \mathbf{v_i})$$

Algorithm 1 Skip-gram with negative sampling, SGD updates in one training window.

Target word x, context window size N, context words $y_0, y_1, \ldots, y_{N-1}$, K negative samples, SGD learning rate α.

1: **for** $i = 0$ to $N-1$ **do**
2:     **for** $j = 0$ to K **do**
3:         **if** $j = 0$ **then**
4:             $s \leftarrow x$, $l \leftarrow 1$
5:         **else**
6:             $s \leftarrow$ rand. neg. sample, $l \leftarrow 0$
7:         $e \leftarrow l - \mathrm{sig}(\mathbf{v_{in,y_i}}^T \mathbf{v_{out,s}})$
8:         $\mathbf{d_{out}} \leftarrow e\mathbf{v_{out,s}}$
9:         $\mathbf{d_{in}} \leftarrow e\mathbf{v_{in,y_i}}$
10:         $\mathbf{v_{out,s}} \leftarrow \mathbf{v_{out,s}} + \alpha\mathbf{d_{in}}$
11:         $\mathbf{v_{in,y_i}} \leftarrow \mathbf{v_{in,y_i}} + \alpha\mathbf{d_{out}}$

Here, $\mathbf{v_x}$ is the word representation of x in M_{in}. $\mathbf{v_y}$ and $\mathbf{v_i}$ are the weight vectors for the target word y and the negative samples in M_{out}. From this equation, we can see that the probability calculation involves several memory-bound vector-vector products. The sigmoid function sig is defined as $\mathrm{sig}(u) = \frac{1}{1+\exp(-u)}$.

Positive and negative samples are processed using the Stochastic Gradient Descent (SGD) algorithm during training. Word vectors in the input and output layers are updated with the objective of maximizing $P(y|x)$ for positive samples and $1 - P(y|x)$ for negative samples. Each word in the training data is processed successively during the training process and the training involves many passes to improve accuracy. In the multithreaded setting, data are partitioned among threads and each thread asynchronously performs updates using target words in its partition. This lock-free scheme is known as the HOGWILD! approach and is frequently used in many tasks that rely on SGD.

In summary, the SGNS model learns a D-dimensional vector representation of each word present in a large text corpus. This is done by using each target word to predict the surrounding N context words in a sliding window manner. Algorithm 1 explains the steps involved in the learning process for one target word x and the N context words surrounding the target word. For each context word y_i, K negative samples are randomly selected. Lines 7-11 correspond to the SGD computation to update the word vectors. All target words are processed in this manner and the entire dataset is processed I times.

2.2 Matrix multiplication throughput

The floating-point throughput of SGNS can be significantly improved by converting vector-based computations into matrix-based computations [6, 12]. However, the performance improvement depends on the sizes of matrices, which in turn depend on the values of K, D, and N.

Fig. 1 gives the single precision generalized matrix multiplication (SGEMM) throughput in Giga FLoating point Operations Per Second (GFLOPS) for input matrices of sizes $(K+1, D)$ and $(D, 2N)$. These are sizes of matrices multiplied in Ji et al.'s pWord2Vec implementation [12]. The matrix multiplications are performed on a

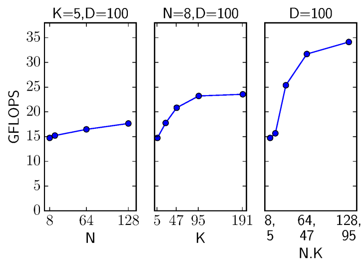


Figure 1: Floating point throughput for $A_{K+1,D} \times B_{D,2N}$.

single core of a 2.4 GHz Intel Xeon E5-2695 v2 (Ivy Bridge) processor. The peak throughput is 38.4 GFLOPS per core. We used Intel Math Kernel Library version 2017.2.174 for this experiment. The title of each plot in Fig. 1 gives the fixed parameter settings and the X axis indicates the varying parameters. The leftmost point in each plot corresponds to a typically used set of values ($K = 5$, $N = 8$ and $D = 100$). We can see that these values result in less than half the peak floating point performance, and hence using larger matrices is necessary for higher floating point throughput. Increasing one parameter (K or N) while keeping the other two fixed results in a slight throughput increase, whereas increasing both K and N while keeping D fixed improves the throughput of SGEMM significantly.

3 RELATED WORK

In this section, we discuss the state-of-the-art parallel implementations of Word2Vec and then describe commonly used test datasets.

We begin by describing shared-memory parallel implementations that are most relevant to our proposed approach. Our work is closely related to Ji et al.'s pWord2Vec implementation [12], which is an extension of Word2Vec with the *negative sample sharing* optimization. This approach facilitates matrix-matrix multiplications instead of vector-vector operations. We present accuracy and training time comparisons of our new approach with Ji et al.'s pWord2Vec. pWord2Vec involves reusing a common set of negative samples in addition to using the same positive sample (target word) for all the words in a window of size N, and this data reuse leads to the computations being expressed as matrix-matrix multiplications.

For example, in Fig. 2a, *bright* and *sunny* are the context words surrounding the target word *day*. Fig. 2b shows the training samples generated for this window in the original Word2Vec implementation. Apart from the target word (*day*) being used as the positive training sample for both the context words, one negative sample is randomly selected for each word. The resulting vector-vector products are listed in Fig. 2b. Fig. 2c shows how multiple vector-vector products of an input context are combined into a single matrix-matrix multiplication step by pWord2Vec. The key change is that both context words use the same word (*desert*) as the negative sample, in addition to sharing the target word. Hence, the word representations of the input words can be multiplied with the shared positive and negative samples using a matrix-matrix multiplication, as shown in Fig. 2c.

BIDMach [6] is a machine learning library based on *rooflined design* principle. The SGNS model of Word2Vec implemented in

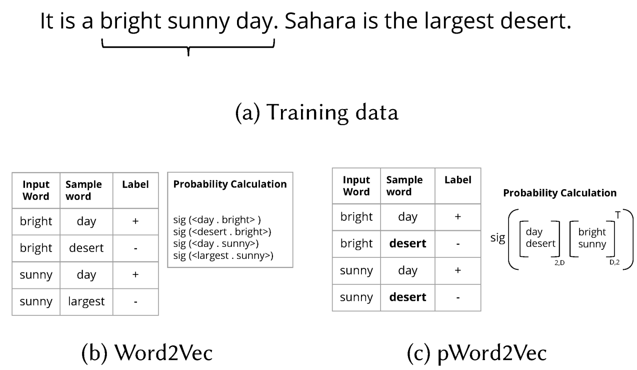


Figure 2: Illustrating the pWord2Vec approach.

BIDMach uses rooflined matrix primitives instead of vector products, while also achieving memory rooflining by merging accesses to the same memory location and by using the large GPU register memory to cache word vectors. However, since this implementation processes one sample at a time, the operations performed are matrix-vector products, as noted by Ji et al. [12]. Ji et al. compare the throughput of their implementation with BIDMach SGNS, and report that their pWord2Vec performs better. This is attributed to the use of matrix-matrix multiplications instead of matrix-vector products.

Vuurens et al. [27] observe that cache collisions due to multi-threaded execution is a major performance bottleneck in the hierarchical softmax variant of Word2Vec. They obtain a 4× improvement over the multithreaded code by a straightforward caching of commonly used word vectors. TensorFlow [1] is a popular machine learning framework, and includes a multithreaded implementation of Word2Vec.

There are several distributed-memory implementations of Word2Vec. Ji et al.'s pWord2Vec [12], Deeplearning4j [25], MLLib [18] replicate the word representations for the entire vocabulary on all tasks. The tasks update their local vectors. The local updates are propagated to other tasks periodically. This replication of word vectors limits the above-mentioned approaches to low-to-medium vocabulary sizes. Stergiou et al. [24] propose novel distributed algorithms which enable processing corpora of more than a trillion words. Their method involves partitioning word vectors using the 128-bit md5sum of the words, constructing a graph representation using the samples generated and processing the graph using a distributed graph processing framework.

Word2Vec's accuracy is often evaluated using word similarity and word analogy tests. Several test datasets are publicly available. These contain manually assigned similarity scores for word similarity tests, as well as word analogy questions and answers. Some test datasets for evaluating word similarity include WordSim353 [10], MEN [5], Mechanical Turk [23], Rare words [17] and SimLex-999 [11]. Two commonly used datasets for word analogy tests include the MSR [20] and Google's analogy datasets [19]. In our evaluations, we use WordSim353 for word similarity tests and Google's dataset for word analogy tests.

Target word	It	is	a	bright	Sunny	day	Sahara	the	largest	desert
Offsets	0	1,7	2	3	4	5	6	8	9	10

Figure 3: Creating inverse index as a preprocessing step.

4 OUR APPROACH

We propose to improve the throughput of Word2Vec by sharing positive/negative samples across multiple context windows, and thus processing multiple contexts simultaneously. This sharing mechanism not only enables us to adopt efficient matrix-matrix multiplications instead of vector-vector products, but also creates large matrices for matrix-matrix multiplications, thus further improving floating point throughput. We call this new optimization *context combining*.

Given a dataset, our approach reads and processes T words at a time in every iteration, and each word is used as a target word for learning. For a target word, the algorithm selects K negative samples and then chooses $C-1$ unprocessed context windows based on these negative samples and the target word. These C context windows (the target word's context window and $C-1$ selected unprocessed ones) are then processed simultaneously for learning. Algorithm 2 describes the proposed algorithm for processing T words of the training data. In the following subsections, we describe each step of the proposed algorithm. These include 1) Preprocessing, where we create an index for identifying unprocessed context windows, and ensuring each word is used as a target word for learning, 2) Identifying related windows, where we select C context windows based on a target word and K negative samples, and 3) SGD update, where we update word representations by processing C context windows simultaneously.

4.1 Preprocessing

This step corresponds to Line 2 in Algorithm 2. After reading T target words into the array *ts*, we create an inverse index mapping each target word to the set of positions where it occurs within *ts*. For the example training data chunk with 11 words ($T = 11$) shown in Fig. 2a, the corresponding inverse index is shown in Fig. 3. This index is constructed to directly identify the positions of a given word within the training data. For example, the word *is* occurs at offsets 1 and 7 within the training data chunk. Since each word occurs at different frequency, storing a separate list of offsets for each word would result in memory wastage if uniform length lists are used or would incur dynamic memory allocation overhead if variable length lists are used. To overcome this, we use Compressed Sparse Row (CSR) representation to store the index which requires $O(V + T)$ memory, V being the vocabulary size.

4.2 Identifying C related windows

Once the indexing is done, all target words are first marked as unprocessed (lines 4-5). The *for* loop in line 6 iterates over each word and considers unprocessed target words ts_i for which $p_i = 0$. In lines 9-11 of Algorithm 2, we copy the unprocessed target word, the corresponding window offset, and the N context words surrounding the target word into local buffers. L is a matrix containing labels indicating the positive and negative samples for each context word. The entry $L_{i,j}$ is 1 if out_i is the positive sample for the context word in_j, and 0 otherwise. Lines 17-18 select K random negative samples

Algorithm 2 Our SGNS approach with the context combining optimization. We use index notation [,] to denote elements within matrices M_{in} and M_{out}. We use subscripts to index all other matrices and vectors.

1: **procedure** pSGNScc
  Input: Training data segment ts containing T words, Context size N, K negative samples, Dimension D, Windows to combine C, and learning rate α.
2: Index (ts)
3: $m \leftarrow CN$
4: **for** $i = 1$ *to* T **do**
5: $p_i \leftarrow 0$ ▷ p tracks processed words
6: **for** $i = 1$ *to* T **do**
7: **if** $p_i \neq 0$ **then** ▷ Skip processed target words
8: continue
9: $out_1 \leftarrow ts_i$
10: $w_1 \leftarrow i$ ▷ Offset of out_1 in ts (window offset)
11: $[in_1, \ldots, in_N] \leftarrow$ context words $\in w_1$ ▷ Select context words
12: **for** $i = 1$ *to* $K+1$ **do**
13: **for** $j = 1$ *to* m **do**
14: $L_{i,j} \leftarrow 0$ ▷ Initialize labels
15: **for** $i = 1$ *to* N **do**
16: $L_{1,i} \leftarrow 1$
17: **for** $i = 1$ *to* K **do**
18: $out_{i+1} \leftarrow$ rand. neg. sample
19: $[w_2, \ldots, w_C] \leftarrow$ FindRel ($W-1$, out) ▷ Find $C-1$ related window offsets
20: **for** $i = 2$ *to* C **do**
21: $[in_{N(i-1)+1}, \ldots, in_{Ni}] \leftarrow$ context words $\in w_i$
22: $outIdx \leftarrow j$ such that $out_j = ts_{w_i}$
23: **for** $j = (i-1)N+1$ *to* iN **do**
24: $L_{outIdx,j} \leftarrow 1$
25: SGDUpdate ($M_{in}, M_{out}, in, out, L, m, K+1, D, \alpha$)
26: **for** $i = 1$ *to* C **do**
27: $p_{w_i} \leftarrow 1$ ▷ Mark target words as processed

1: **procedure** SGDUpdate($M_{in}, M_{out}, in, out, L, N, K, D, \alpha$)
2: **for** $i = 0$ *to* $N-1$ **do**
3: $inM_{i,*} \leftarrow M_{in}[in_i, *]$
4: **for** $i = 0$ *to* $K-1$ **do**
5: $outM_{i,*} \leftarrow M_{out}[out_i, *]$
6: $E \leftarrow L - sig\ (MatMul\ (outM, inM^T))$
7: $D_{in} \leftarrow MatMul\ (E, inM)$
8: $D_{out} \leftarrow MatMul\ (E^T, outM)$
9: **for** $i = 0$ *to* $K-1$ **do**
10: $M_{out}[out_i, *] \leftarrow M_{out}[out_i, *] + \alpha\ D_{in_{i,*}}$
11: **for** $i = 0$ *to* $N-1$ **do**
12: $M_{in}[in_i, *] \leftarrow M_{in}[in_i, *] + \alpha\ D_{out_{i,*}}$

from the vocabulary. While pWord2Vec selects one sliding window of N words at a time and processes the window using $K + 1$ sample words, our approach selects up to C windows from training data and reuses the $K + 1$ samples to process all these windows. In Line 19, the algorithm finds up to $C - 1$ additional windows that contain any of the sample words using the inverse index created in the preprocessing step.

We illustrate context combining in Fig. 4. Here, *day* is the unprocessed target word and *desert* is the negative sample. In this figure, *C2* is the additional window selected because it contains the negative sample *desert*, which becomes the positive sample for *C2* and the remaining sample *day* is assigned as the negative sample for *C2*. As a result of combining two contexts, the context matrix used for probability calculation contains three word vectors instead of the two word vectors in the pWord2Vec approach shown in Fig. 2c. The larger matrices resulting from context combining yield a higher matrix multiplication throughput.

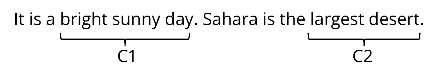
Using the inverse index to find related windows incurs additional overhead due to random accesses to the *Index* and *wOffset* arrays. If the size of this index is large, this overhead would be higher due to potential random accesses to these arrays and the resulting cache and TLB misses. Since the size of the *wOffset* array depends upon T, a small T value is desirable to reduce this access overhead. However, very small T values would result in reading only a few target words at a time. It would not be possible to find multiple windows containing the sample words, thus resulting in no throughput improvement. An optimal value of T can be found by experimentation with a subset of the training data.

There are multiple approaches to select C windows for context combining. Since our approach statically assigns negative samples for the related windows, we find that processing all windows for a given target word at the same time reduces the effectiveness of the learning process because of very few negative samples used to process all input words for that target word. So, instead, we select related windows in a round-robin fashion. For example, if $w_0, \ldots, w_k$ are the sample words, we first find a window whose target word is w_0, then w_1, and so on, and then back to w_0, until we find C windows or there are no more related windows. The number of windows combined is close to C during the initial iterations of the *for* loop in Line 6 and reduces during the later iterations, as more and more windows are processed, which results in fewer than C unprocessed windows for a given set of samples. We show that with this approach, the effectiveness of learning is comparable to pWord2Vec.

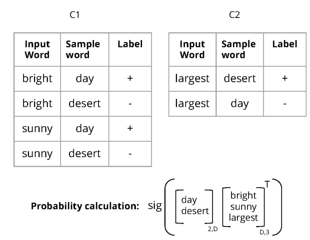
4.3 Perform SGD update

After selecting $C - 1$ additional windows to combine, the *for* loop in line 20 of Algorithm 2 aggregates the context words for the $C - 1$ windows and sets the labels for each of these words. Then SGD is performed on these windows simultaneously to update the word vectors corresponding to the m context and $K + 1$ sample words. This step corresponds to Line 25 in Algorithm 2 and is described in the procedure SGDUpdate.

This procedure begins by copying the input word representations of the context words into matrix *inM* and the output representations of the samples into matrix *outM*. The subsequent steps are similar to Algorithm 1 except for the fact that this procedure operates on matrices instead of vectors. The procedure calculates probabilities, errors, and gradients, and finally the word representations are updated. For example, in Fig. 4, input vectors of C1 (*bright,sunny*) and C2 (*largest*) are copied into *inM* and the output vectors of sample words *desert, day* are copied into the *outM* matrix. After SGD computations, the target words of these windows are marked as processed to avoid processing again in line 27 of Algorithm 2.



(a) Training data segment (T=11)



(b) Context combining approach

Figure 4: Sample selection in our method ($K = 1$, $N = 2$).

4.4 Parallelization

Algorithm 2 describes the sequential processing of target words within a training data segment containing T words. The parallelization strategy of our approach is similar to the original Word2Vec: the training data segments are divided equally among threads and each thread performs SGD computations and updates using the target words present in its data segments. Updates of multiple threads proceed asynchronously ignoring race conditions. This static partitioning of training data leads to a good load balance in practice. Our context combining approach differs from Word2Vec since we may process up to C windows at a time, resulting in a higher throughput.

5 EXPERIMENTS AND RESULTS

5.1 Experimental setup

5.1.1 Hardware. We primarily use a single compute node of the Stampede supercomputer at the Texas Advanced Computing Center (TACC). Each compute node of Stampede has two 8-core Intel Xeon E5-2680 (Sandy Bridge) processors and an Intel Xeon Phi (Knights Corner) coprocessor with 32 GB DDR3 memory. We do not use the coprocessor in this work. In addition, we evaluate performance on a single node of the new Stampede2 supercomputer, which has Intel Xeon Phi 7250 Knights Landing (KNL) processors. These processors have 68 cores with 4-way simultaneous multithreading, and thus support 272 hardware thread contexts. Each node has 96 GB DDR4 and 16 GB high-speed MCDRAM memory. The MCDRAM memory is used as an L3 cache in our experiments.

Table 1: The default settings for hyper-parameters used.

	Hyper-parameter	Value	
		text8	1B
I	Number of epochs/iterations	10	5
D	Vector size	100	300
N	Max window size	8	5
K	Number of neg. samples	5	
T	Data segment size	500K	
C	Contexts combined	8	

5.1.2 Data and queries. We use two training datasets in our evaluations. *text8* has approximately 17 million words taken from Wikipedia, with a vocabulary size of 71,292 unique words. The *One Billion Word* benchmark (1B) [7] dataset contains 805 million words of news crawl data. The vocabulary size is 1.1 million.

To evaluate performance on the word similarity task, we use the WordSim353 (ws353) [10] benchmark queries containing 353 word pairs with human assigned similarity scores. To evaluate word analogy performance, we use the Google analogy queries [19], which have 19,544 questions of the form *athens is to greece as baghdad is to ?*.

We use the evaluation methods described by Levy et al. [16]. Effectiveness of a model on word similarity tests is evaluated by ranking the word pairs based on the cosine similarity of the word vectors and then measuring the Spearman's correlation with the ratings present in the test dataset. Effectiveness of a model on word analogy tests is evaluated as follows. For a given word analogy question of the form *a is to a^* as b is to b^**, where the answer b^* is hidden, the word that maximizes the cosine similarity function (Equation 1) is taken to be the answer of the model. The answer is correct if it is same as b^*. Model accuracy is reported as the fraction of questions answered correctly.

$$cos(b^*, a^* - a + b) = cos(b^*, a^*) - cos(b^*, a) + cos(b^*, b) \quad (1)$$

5.1.3 Compile and Run time configurations. We compare our context combining approach (denoted *pSGNScc*) to Mikolov et al.'s Word2Vec and Ji et al.'s pWord2Vec approaches. We report training time per *epoch* (or one pass over training data) and the accuracy on word similarity and analogy tests. We used Intel compiler version 15.0.2 on Stampede and version 17.0.4 on Stampede2. We pin threads to cores for all three methods. Table 1 gives the default settings we use for the experimental hyper-parameters. Unless explicitly stated, all experiments are run on a single compute node of Stampede supercomputer using 16 threads.

5.2 Comparisons with state-of-the-art implementations

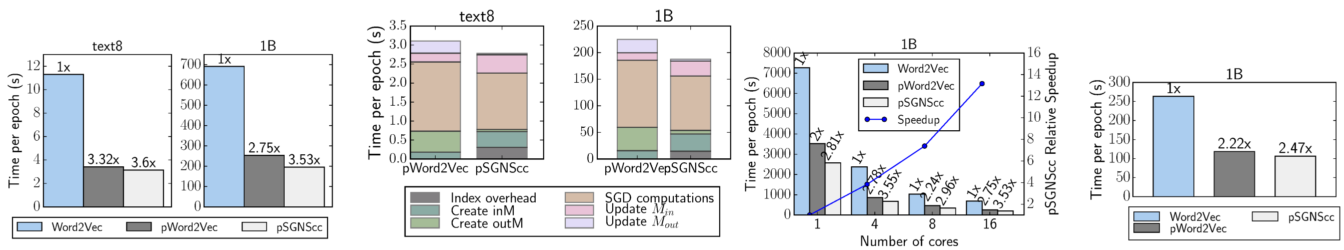
5.2.1 Parallel Performance Evaluation. Fig. 5a compares the training time with the three methods on a single node of the Stampede supercomputer. For text8, pSGNScc results in a 3.6× speedup over Word2Vec and a 1.08× speedup over pWord2Vec. For the 1B dataset, the speedup is 3.53× over Word2Vec and 1.28× over pWord2Vec. The increase in speedup over pWord2Vec from 1.08× (text8) to 1.28× (1B) is due to the following reason. As the vocabulary size increases from 71K (text8) to 1.1M (1B), selecting negative samples becomes more expensive, since there are possibly more random accesses to a larger vocabulary buffer in memory, leading to more last level cache misses. As pSGNScc method uses fewer negative samples per target word compared to the pWord2Vec approach, the performance improvement is more pronounced on larger datasets.

We analyze the time taken by each step of pSGNScc and pWord2Vec in order to understand the reason for the performance improvement. The step-wise breakdown of time for one epoch of the two methods are shown in Fig. 5b. *Index overhead* refers to the time taken for creating the inverse index and traversing this index to find related windows during training process. The *Create inM* and *Create outM* steps refer to copying context word vectors from M_{in} to the *inM* matrix, and target word vectors from M_{out} to *outM* before SGD computations. *SGD computations* refers to probability and gradient computations, and *Update M_{in}* and *Update M_{out}* are the steps for updating the M_{in} and M_{out} matrices after computation.

Fig. 5b shows that in pSGNScc, *Create inM* and *Update M_{in}* are slower than the pWord2Vec approach, whereas *SGD computation*, *Create outM* and *Update M_{out}* steps are faster, resulting in an overall improvement in throughput. The time for creating the *inM* matrix increases because the pWord2Vec approach processes consecutive windows. Hence, it has better cache locality. pSGNScc selects related windows from random offsets in the training data, hence resulting in poor cache locality. This is the reason for increase in time for updating M_{in} matrix as well. *outM* matrix creation time and M_{out} matrix update time decreases in pSGNScc because we use $1/C$ fewer negative samples. SGD computation is faster because of the larger matrices created, which results in better matrix-matrix multiplication throughput.

Fig. 5c compares the training time of pSGNScc with Word2Vec and pWord2Vec for a varying number of cores. We use the 1B dataset for this experiment. Our implementation uses the default hyper-parameter settings and hence, identifies a maximum of 8 windows and processes them together during the training process. As expected, pWord2Vec and pSGNScc are much faster than the original Word2Vec implementation. pSGNScc is faster than the pWord2Vec approach because it creates larger matrices for multiplication while reducing overheads involved. The training time for Word2Vec, pWord2Vec, and pSGNScc methods for the 16-thread run are 691, 251, and 196 seconds, respectively. By processing 8 windows at a time, pSGNScc achieves a 3.53× speedup over Word2Vec and a 1.28× speedup over pWord2Vec. The speedup obtained by 16 threaded execution of each method over single threaded execution of the same method are 10.5× (Word2Vec), 14× (pWord2Vec) and 13.6× (pSGNScc). The speedup for the original implementation is less due to cache collisions and memory boundedness, whereas the remaining two methods show good scalability characteristics. These two methods show near-linear scalability up to 8 cores (7.64× and 7.4×), after which remote NUMA accesses limit scaling.

Fig. 5d compares the training time on a single node of Stampede2 supercomputer, which contains Intel Knights Landing (KNL) processor. For the 1B dataset, pSGNScc results in a 2.47× speedup over Word2Vec and 1.11× speedup over pWord2Vec. These values are less than the speedups achieved on Stampede system because



(a) 16-core performance (b) Breakdown of time (c) Parallel scaling (d) KNL performance

Figure 5: Parallel performance results and comparisons.

Table 2: Comparing accuracy of the three implementations on Similarity and Analogy queries. Higher values are better.

Method	Similarity		Analogy	
	text8	1B	text8	1B
Word2Vec	.671	.636	.297	.330
pWord2Vec	.682	.639	.309	.327
pSGNScc	.685	.633	.322	.328

in the Stampede2 runs, we used hyperthreading and hence a core's resources such as cache, vector and floating point units are shared by 4 threads instead of a single thread. This leads to better resource utilization for the pWord2Vec approach and hence the reduced speedup for pSGNScc on KNL processor. Improving the performance of pSGNScc on KNL is a part of our future work.

5.2.2 Evaluating accuracy. We evaluate the effectiveness of the learning process of pSGNScc using word similarity and word analogy tests and present results comparing pSGNScc's accuracy with Word2Vec and pWord2Vec. As mentioned previously, we use WordSim353 and Google analogy datasets for measuring accuracy on word similarity and word analogy tests respectively. In pSGNScc, once we identify an unprocessed target word and select K random negative samples, we identify $C - 1$ related windows. For these windows we statically assign negative samples. Table 2 shows that pSGNScc and pWord2Vec outperform Word2Vec in terms of accuracy for text8 dataset. This could be due to shared negative samples for the context words in both the methods. For 1B dataset, the accuracy of all three methods are similar.

5.3 Tuning T and C

The number of target words read in a segment (T) and the maximum number of contexts to combine (C) are two new hyper-parameters introduced in the pSGNScc approach. The index is created on a T-word training data segment. As mentioned in Section 4.2, a small value of T results in fewer related windows processed simultaneously and a large value of T increases random access overhead during inverse index lookups. Hence, finding the appropriate value of T is essential for achieving high throughput. Table 3 shows the time per epoch, the overhead due to index creation and lookup, and the number of related windows processed simultaneously on average when varying T.

When T is 100K, pSGNScc can only process 5.49 related windows simultaneously on average, and hence the training time is higher than when using larger T values, such as 500K or 1M. The index time includes both creation and lookup time. Index creation time is highest when T is 100K, because a small T value increases the number of times the index is created for the entire dataset. Index lookup time is low when T is 100K, because index lookup involves random accesses to small index arrays. Hence, index creation time decreases with increase in T and index lookup time increases with increase in T. This results in least overhead when T is 500K. An appropriate way to select T would be selecting the smallest T which results in identifying close to W related windows on average. Hence, we use 500K as the default setting in our experiments.

Similar to choosing T, choosing an appropriate C value also involves trade-offs. As C increases, our approach combines more windows, resulting in higher throughput until a point where C related windows do not exist in the training data segment of size T, or the overhead in identifying related windows negates improvement in performance. Also, from the leftmost plot in Fig. 1 it can be seen that for a fixed K and D parameters, increasing N does not improve matrix-matrix multiplication throughput beyond certain point. Considering all these limiting factors, we select a C value that maximizes the training throughput by experimenting on a subset of training data.

Table 3 shows that as C increases, the index overhead increases, because identifying more related windows involves more lookups in the inverse index. Also, the candidate window positions obtained from the index lookup are checked to ensure that they have not been processed already. Both these lookups incur more random memory accesses, resulting in the increase in overhead. SGD computation time continues to decrease until $C = 16$. However, the improvement is less significant for $C > 8$. We have used $C = 8$ as the default setting in our experiments.

6 CONCLUSIONS

In this work, we propose a new optimization called context combining to improve throughput of Word2Vec's SGNS algorithm. Our approach results in a higher floating point throughput by performing matrix-matrix multiplications on large matrices, while reducing the overhead involved in creating and updating the matrices by

Table 3: Performance impact of T and C on the 1B dataset.

	Value of T			
	100K	500K	1M	2M
Time per epoch (s)	220.13	**196.03**	196.74	201.13
Index time (s)	17.20	**14.85**	17.97	24.22
Avg. related windows (≤ 8)	5.49	7.53	7.80	**7.92**
	Value of C			
	1	4	8	16
Time per epoch (s)	258.38	208.45	196.03	**191.08**
Index time (s)	**3.16**	12.47	14.92	20.25
SGD Computations (s)	129.94	108.27	102.12	**95.69**

sharing samples across multiple windows. We demonstrate a 3.53× speedup in training time compared to the original Word2Vec SGNS, and a 1.28× speedup compared to Ji et al.'s pWord2Vec implementation. The training accuracy is comparable to these approaches.

Although matrix multiplication throughput is higher in comparison with pWord2Vec, it is still small compared to the system's peak floating point throughput since we increase the size of only one of the matrices (containing context words) involved in the multiplication. In other words, there exists much room for improvement. In the future, we would like to explore variants of this approach to further improve the throughput, such as combining unrelated contexts (contexts that do not share words), or combining successive contexts (contexts that share the most words). Another direction for our future work is to extend the context combining strategy to GPU and distributed-memory settings.

ACKNOWLEDGMENT

This research is supported in part by the US National Science Foundation grants ACI-1253881, CCF-1439057, IIS-1717084, and SMA-1360205. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) [26], which is supported by National Science Foundation grant number ACI-1548562.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Google Code Archive. 2013. word2vec: Tool for computing continuous distributed representations of words. (2013). https://code.google.com/archive/p/word2vec/.

[3] Ehsaneddin Asgari and Mohammad RK Mofrad. 2015. Continuous Distributed Representation of Biological Sequences for Deep Proteomics and Genomics. *PLoS ONE* 10, 11 (2015), e0141287.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research* 3, Feb (2003), 1137–1155.

[5] Elia Bruni, Gemma Boleda, Marco Baroni, and Nam-Khanh Tran. 2012. Distributional Semantics in Technicolor. In *Proc. Annual Meeting of the Association for Computational Linguistics (ACL)*.

[6] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. 2015. Machine Learning at the Limit. In *Proc. Int'l. Conf. on Big Data (Big Data)*.

[7] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. *One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling.* Technical Report. Google. http://arxiv.org/abs/1312.3005.

[8] Kenneth Ward Church and Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational linguistics* 16, 1 (1990), 22–29.

[9] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proc. Int'l. Conf. on Machine Learning (ICML)*.

[10] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing Search in Context: The Concept Revisited. In *Proc. Int'l. Conf. on World Wide Web (WWW)*.

[11] Felix Hill, Roi Reichart, and Anna Korhonen. 2015. SimLex-999: Evaluating Semantic Models with (Genuine) Similarity Estimation. *Computational Linguistics* 41, 4 (2015), 665–695.

[12] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. 2016. Parallelizing Word2Vec in Multi-Core and Many-Core Architectures. In *Proc. Int'l. Workshop on Efficient Methods for Deep Neural Networks (EMDNN)*.

[13] Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-thought vectors. In *Proc. Conf. on Neural Information Processing Systems (NIPS)*.

[14] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From Word Embeddings To Document Distances. In *Proc. Int'l. Conf. on Machine Learning (ICML)*.

[15] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural Architectures for Named Entity Recognition. In *Proc. Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT)*.

[16] Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving Distributional Similarity with Lessons Learned from Word Embeddings. *Transactions of the Association for Computational Linguistics* 3 (2015), 211–225.

[17] Thang Luong, Richard Socher, and Christopher D Manning. 2013. Better word representations with recursive neural networks for morphology. In *Proc. Conf. on Computational Natural Language Learning (CoNLL)*.

[18] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proc. Int'l. Conf. on Learning Representations (ICLR) Workshop*.

[20] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic regularities in continuous space word representations. In *Proc. Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT)*.

[21] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proc. Conf. on Neural Information Processing Systems (NIPS)*.

[22] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*.

[23] Kira Radinsky, Eugene Agichtein, Evgeniy Gabrilovich, and Shaul Markovitch. 2011. A Word at a Time: Computing Word Relatedness using Temporal Semantic Analysis. In *Proc. Int'l. Conf. on World Wide Web (WWW)*.

[24] Stergios Stergiou, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis. 2017. Distributed Negative Sampling for Word Embeddings. In *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*.

[25] Deeplearning4j Development Team. 2017. Deeplearning4j: Open-source distributed deep learning for the JVM. (2017). Apache Software Foundation License 2.0, http://deeplearning4j.org.

[26] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering* 16, 5 (2014), 62–74.

[27] Jeroen BP Vuurens, Carsten Eickhoff, and Arjen P de Vries. 2016. Efficient Parallel Learning of Word2Vec. In *Proc. Int'l. Conf. on Machine Learning (ICML) ML Systems Workshop*.

[28] Will Y Zou, Richard Socher, Daniel M Cer, and Christopher D Manning. 2013. Bilingual Word Embeddings for Phrase-Based Machine Translation. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*.

A ARTIFACT DESCRIPTION

A.1 Abstract

Our artifact contains source files for three implementations of Word2Vec compared in this paper namely, Word2Vec, pWord2Vec and pSGNScc. We have obtained Word2Vec source file from https://github.com/dav/word2vec (commit 80be14a) and pWord2Vec source file from https://github.com/IntelLabs/pWord2Vec (commit e6d0d1e). We have modified both these source files to instrument time for different training stages. We have implemented our Context Combining method (pSGNScc) on top of pWord2Vec. pWord2Vec and pSGNScc implementations require Intel C++ compiler and MKL library. The three implementations have been tested on shared memory systems with multicore Intel CPUs.

We have provided Collective Knowledge (CK) integration to automatically check for dependencies, compile, run experiments presented in our paper and present execution time and accuracy results in tabular format.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithm:** Context Combining (CC).
- **Program:** pSGNScc.cpp implements CC algorithm. Also included are word2vec.c and pWord2Vec.cpp programs for comparison.
- **Compilation:** Intel Compiler.
- **Binary:** Binary not included.
- **Data set:** The 2 datasets used in this paper namely, text8 and 1B have been added as CK packages and can be downloaded via CK. text8 dataset requires ≈100MB and 1B dataset requires ≈6GB disk space.
- **Run-time environment:** Our artifact has been developed and tested on Linux environment. The main software dependencies include Intel C++ compiler, MKL library and Python version > 2.7.
- **Hardware:** We used Intel Xeon E5-2680 (Sandy Bridge) and Intel Xeon Phi (Knights Landing) processors for experiments presented in our original paper. Similar hardware should result in similar speedup results. Main memory usage is ≈5GB.
- **Output:** All the three programs output a text file containing the learnt word representations. These files are used to evaluate training accuracy. These files will be created in the user's home directory and hence home directory should have at least 4GB free disk space. The experiments output a table to the console. Each table corresponds to one Figure in the paper indicating execution times and accuracy.
- **Experiment workflow:** We have used CK framework to create experiment workflows.
- **Publicly available?:** Yes.

A.2.2 How delivered. Our artifact is available on GitHub: https://github.com/vasupsu/IA3_Paper16_ArtifactEvaluation.

A.2.3 Hardware dependencies. Our experiment workflows use 1 to 16 threads to report training time and hence we suggest a machine with at least 16 CPU cores and 5 GB main memory.

A.2.4 Software dependencies. We recommend Linux x86_64 environment for running the experiments. We require Intel C++ compiler and hyperwords [16] package for evaluating training accuracy. Hyperwords package has been added as a run time dependency and will be downloaded when the programs are run for the first time.

A.2.5 Datasets. The datasets text8 and 1B have been added as CK packages and can be downloaded via CK.

A.3 Installation

CK can be installed by cloning a development CK version from GitHub and then setting the PATH environment variable.

```
$ git clone https://github.com/ctuning/ck.
    git ck-master
$ export PATH=$PWD/ck-master/bin:$PATH
```

Our artifact named IA3_Paper16_ArtifactEvaluation can be installed via CK as:

```
$ ck pull repo --url=https://github.com/
     vasupsu/IA3_Paper16_ArtifactEvaluation
```

Datasets can be downloaded and installed by running the command

```
$ ck install package --tags=dataset,words
```

twice and selecting 1 dataset each time.

A.4 Experiment workflow

We have provided 4 experiment workflows implemented as CK modules in the IA3_Paper16_ArtifactEvaluation repository. These modules are named *ia3-2017-paper16-figure5a*, *ia3-2017-paper16-table2*, *ia3-2017-paper16-table3a* and *ia3-2017-paper16-table3b*. The module ia3-2017-paper16-figure5a outputs tables to verify results in Figures 5a, 5b and 5c. The module ia3-2017-paper16-table2 is used to verify results in Table 2 and the modules ia3-2017-paper16-table3a, ia3-2017-paper16-table3b are used to verify results in Table 3.

Each of the experiment workflows can be run using the following CK command:

```
$ ck run_expt <module-name>
```

A.5 Evaluation and expected result

The training times output by the experiment workflows need not match exactly with the numbers reported in the paper owing to disparity in CPU clock frequency and concurrency settings. However, the relative speedup of pSGNScc method over pWord2Vec and Word2Vec and the accuracy of different methods obtained from the experiment outputs should be similar to the values reported in the paper. The relative speedup information can be obtained using the output table of experiment *ia3-2017-paper16-figure5a* which corresponds to Figure 5a.

A.6 Experiment customization

Apart from the experiment workflows provided as CK modules, each program can be run independently with different hyperparameter settings. The CK program names corresponding to Word2Vec, pWord2Vec and pSGNScc implementations are *word2vec*, *pword2vec* and *pSGNScc* respectively. For example, the CK command to run *pSGNScc* program using 16 threads is:

```
$ ck run program:pSGNScc --env.CK_THREADS=16
```