Towards Parallel Implementation of Associative Inference for Cogent Confabulation

Zhe Li, Qinru Qiu
Dept. of Electrical Engineering & Computer Science
Syracuse University
Syracuse, NY 13224, USA
{zli89, qiqiu}@syr.edu

Mangesh Tamhankar
Intel Corporation
Santa Clara, CA USA
mangesh.tamhankar@intel.com

***Abstract*—The superb efficiency and noise resilience of human cognizance comes from the extensive highly associative memory. For example, it is easy for human to recognize occluded or incomplete text images based on its context. Associative inference in the neocortex system is a concurrent process. Serial implementation of this concurrent process not only hinders its performance, but also limits the quality of recall. This paper investigates parallel implementation of associative inference using cogent confabulation model, which is a highly cross-dependent and cyclic knowledge network that supports probabilistic inference. By breaking the fixed processing order, which is typical in sequential processing, and introducing randomness generated from the race conditions in parallel processing, we do not only reduce the runtime, but also improve the accuracy. Further improvement can be achieved by scheduling the lexicon processing intermittently, which provides time for the changes to settle down. Using sentence construction as a case study, we demonstrate that the parallel implementation provides up to 93.4% reduction in computation time and 5% improvement in recall accuracy.**

***Keywords*—Cogent Confabulation; Sentence Completion; Parallel Programming; Multi-Threading**

I. Introduction

Human perception and cognition involve two steps, sensing and association. The association area is by far the most developed part of the cerebral cortex. The associative inference is related to brain activities in routine, repetitive situations and well-precedent problems. It has been used to explain the perception of sensory input [1] and language learning [2]. The superb of human cognizance comes from its extensive highly associative memory. For example, it is easy for human to correct errors and recover the damages in document images or speech; while this has always been a difficult task for conventional OCR or speech recognition tools.

Many associative memory models have been proposed. They include attractor network associative memories, bidirectional associative memories, and sequence associative memories, etc. In this work, we focus on the cogent confabulation model, which is a belief network with recurrent connections. Cogent confabulation arranges neurons into lexicons. The input of the cogent confabulation is the initial status of the neurons, which corresponding to noisy and ambiguous observations or lack of information. The recall process is an iterative excitation and inhibition among neurons. The feedback connections also loop back the neural activities and gradually refine the memory until at the end only the set of the most highly associated neurons are active. This model has been applied in text image recognition and sentence construction in previous works [3–8].

Associative inference in the neocortex system is a concurrent process, where neurons update their states simultaneously. During this procedure, there is no deterministic order among neurons. In all previous works, cogent confabulation is implemented as a sequential process, where neurons are updated based on a prefixed order. The status of the neurons in one lexicon depends on the output of neurons in some other lexicons. Due to feedback connections, the data dependency is cyclic. None of the fixed evaluation order can satisfy all precedent constrains.

In this work we aim at parallelizing the algorithm on a multicore processor. The parallel framework consists of a thread pool, where each thread evaluates the status of specific lexicons. In addition to reduced computation time, we found that the parallel implementation also improves recall accuracy. The racing among threads breaks the fixed processing order in sequential processing and introduces randomness. The parallel architecture allows neurons to use the most up-to-date information of their neighbors.

The number of lexicons to be processed in a confabulation model is usually much greater than the number of active threads that can be supported by a general purpose CPU. Each thread needs to process multiple lexicons. We design a lexicon scheduling algorithm to further add randomness in the lexicon processing order and at the same time ensure a balanced progress in status update among neurons. Based on the algorithm, a thread switches from one lexicon to another when the neuron activity of the former has reached to a state that is informational.

The contributions of this paper can be summarized as follows:

- A parallel implementation for cogent confabulation is developed using multi-threading and blocking queue techniques.
- We demonstrate that the parallel implementation not only reduces runtime, but also improves recall accuracy by breaking the fixed evaluation order and maintain a balanced progress in status update among all neurons.
- A lexicon scheduling algorithm is presented that provides further improvements.

This work is partially supported by the National Science Foundation under Grant CCF-1337300 and Intel Corporation under CG #20263363.

The experimental results show that up to 93.4% reduction in runtime and 5% increase in recall accuracy can be achieved using the proposed parallel implementation and scheduling algorithm.

The rest of the paper is organized as follows. In Section II we introduce the basics of cogent confabulation model. Section III describes parallelization scheme and its improvements. Section IV implements confabulation for sentence completion problem and compares performance and accuracy of different implementations. Section V summarizes the work.

II. Cogent Confabulation

Inspired by human cognitive process, cogent confabulation [9] mimics human information processing including Hebbian learning, correlation of conceptual symbols and recall action of brain. It represents the observation using a set of features. These features construct the basic dimensions that describe the world of applications. Different observed attributes of a feature are referred as *symbols*, which are analogous to neurons in a typical neural network. The names symbols and neurons are used interchangeably in the rest of the paper. The set of symbols used to describe the same feature forms a *lexicon* and the symbols in a lexicon are exclusive to each other. Symbols in different lexicons excite each other and symbols in the same lexicon inhibit each other. The connections between two lexicons are bidirectional, and data flow in the network is cyclic.

During learning, matrices storing posterior probabilities between neurons of two features are captured. They are referred as the *knowledge links (KLs)*. The (i, j)th entry of a KL, quantified as the conditional probability $P(s_i|t_j)$, represents the Hebbian plasticity of the synapse between ith symbol in source lexicon s and jth symbol in target lexicon t. The knowledge links are constructed during learning by extracting and associating features from the training set and the collection of all knowledge links in the model forms a *knowledge base (KB)*.

The input of the recall function is a set of activated symbols A_l for each lexicon l. These symbols are referred as *candidates*. They correspond to a noisy observation of the target. In this observation, some features are observed with great ambiguity, therefore multiple symbols are activated in the corresponding lexicons, i.e. $|A_l| \geq 1$. In extreme cases, no observation is obtained for certain features, therefore, all symbols in those lexicons are activated as potential candidates. The goal of the recall process is to resolve the ambiguity and select the set of symbols that are most highly associated with each other using the statistical information obtained during the learning. At the end of the recall process, we obtain a refined activation set A_l^*, and $|A_l^*| = 1, \forall l$. This is achieved a sequence of iterative excitation and inhibition among neurons as described in the following.

Each neuron in a target lexicon receives an excitation from neurons of other lexicons through KLs, which is the weighted sum of its incoming excitatory synapses. Let l denote a lexicon, F denote the set of lexicons that have knowledge links going into lexicon l, and S_l denote the set of symbols that belong to lexicon l. The excitation $el(t)$ of a symbol t in lexicon l is calculated by summing up all incoming knowledge links:

$$el(t) = \sum_{k \in F} \{ \sum_{s \in S_k} [I(s) w_{kl} \ln(\frac{P(s|t)}{p_0})] + B \}, t \in S_l \quad (1)$$

$$I(s) = \frac{el(s)}{\sum_{j \in S_k} el(j)}, s \in S_k \quad (2)$$

$I(s)$ is the normalized excitation level across all actives in the same lexicon. The parameter p_0 is the smallest meaningful value of $P(s_i|t_j)$. w_{kl} is the weighting factor which is a linear function of mutual information [10] of KL from lexicon k to lexicon l [6]. The parameter B is a positive global constant called the *bandgap*. The purpose of introducing B in the function is to ensure that a symbol receiving N active knowledge links will always have a higher excitation level than a symbol receiving $(N-1)$ active knowledge links, regardless of their strength. As we can see, the excitation level of a symbol is actually its log-likelihood given the observed attributes in other lexicons.

Among neurons in the same lexicon, those that are least excited will be suppressed and the rest will fire and become excitatory input of other neurons. Their firing strengths are normalized and proportional to their excitation levels. As neurons gradually being suppressed, eventually only the neuron that has the highest excitation remains to fire in each lexicon and the ambiguity is thus resolved.

Algorithm 1: Baseline sequence confabulation recall algorithm

```
Data: The activation set A_l (|A_l| ≥ 1) of symbols in each lexicon l,
      predefined maxAmbiguity, maxIteration
Result: The refined activation set A_l^* (|A_l^*| = 1)
N ← maxAmbiguity
Initialize the set of unknown lexicons L_u = {l : |A_l| > 1}
while N > 1 do
    converged ← false
    iterationCount ← 0
    while ¬converged do
        for each l_u ∈ L_u do
            for each symbol s ∈ A_{l_u} do
                calculate el(s)
            end
            sort(A_{l_u}) based on el(s), ∀s ∈ A_{l_u}
            set the first N symbols in A_{l_u} as active
            for each symbol s ∈ A_{l_u} do
                calculate I(s)
            end
        end
        iterationCount ← iterationCount + 1
        if active symbol set unchanged ∨
        iterationCount ≥ maxIteration then
            converged ← true
        end
    end
    for each l_u ∈ L_u do
        A_{l_u} ← the first N symbols in A_{l_u}
    end
    N ← N − 1
end
A_l^* ← A_l, ∀l
output A_l^*, ∀l
```

Algorithm 1 shows the recall function as a sequential process. The candidates excitation levels are calculated lexicon by lexicon in series. Due to the recurrent connections between

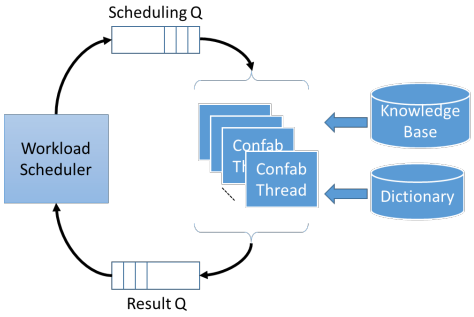

Fig. 1. Intuitive parallel confabulation architecture

lexicons, the evaluation needs to iterate several times to ensure that changes in the excitation level of lexicons propagate through the network before we prune one symbol from every lexicon. It is not optimal to prune all lexicons at the same time, because the decision is made upon the excitation level calculated using the old symbol status. After a symbol in one lexicon is pruned, the excitation levels of symbols in other lexicons are likely to change. Therefore, it is possible that we are going to prune something that should be kept. However, because it is expensive to evaluate the excitation level sequentially, we cannot afford to update the excitation level each time after pruning a symbol in one lexicon. And due to the fixed processing order, some lexicons always have the privilege of making decision on more update-to-date information than the others.

III. Parallel Implementation of Associative Inference

To fully explore the computation power of multi-core multi-thread processors, we investigate parallel implementation of the recall function of cogent confabulation. Starting from a naïve implementation that simply duplicates multiple copies of the recall function and run them in parallel, we improve the architecture by adopting finer grained parallelism and better-controlled scheduling algorithms. In Section IV, the experimental results will show that these additions not only reduce the runtime but also improve the recall accuracy.

A. Request Level Parallelization

The intuitive design of parallelization is to run multiple copies of the recall functions in Algorithm 1 using multiple threads. Each thread processes an independent confabulation recall, and all threads share the same knowledge base. As shown in Fig. 1, for each incoming confabulation request, the workload scheduler will collect symbol candidates to assemble the initial activation set as the input of the recall function. Once the initial activation set is assembled, it will be placed in the scheduling queue. And each confab thread will de-queue one request and run Algorithm 1.

At the end of recall, the refined activation set will be sent to the result queue. Please note that in this design, scheduling queue and result queue are both thread safe blocking queues, which enable an automatic load balancing among multiple threads. This design can simultaneously process as many recall functions as the number of confab threads. However, within a confab thread, the processing is still sequentially. All the previously mentioned limitations of the sequential implementation,

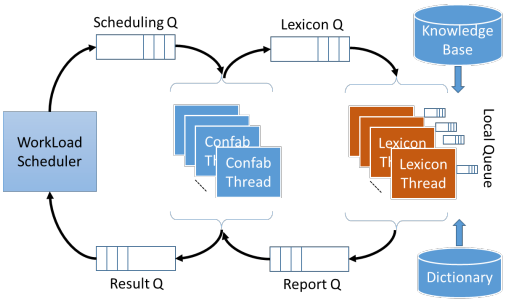

Fig. 2. Parallel confabulation architecture on lexicon level

such as fixed processing order, the need of iterative evaluation, and the possibility of pruning some symbol without sufficient information, still exists in this nave implementation.

B. Lexicon Level Parallelization

To emulate the parallel structure in a biological neural system, we investigate finer grained parallelism where lexicons are processed in parallel. A set of lexicon threads are created. Once a confabulation thread receives a request from the scheduling queue, it divides the input to multiple lexicons. It places those lexicons with ambiguous or unknown information (i.e. the lexicons l_u with more than one active candidates) with their activation set (i.e. A_{l_u})) into a lexicon queue. Each lexicon thread will fetch its input from the lexicon queue, iteratively performs excitation level calculation and pruning the inactive symbols until there is only one active symbol. The lexicon together with the refined activation set is put into a report queue, which will be read by the confabulation thread. After the confabulation thread collected the report for all the lexicons belonging to the same recall request, it will forward the result to the result queue.

The new architecture is depicted in Fig. 2, except the round-robin queues, which are not needed to achieve the lexicon level parallelism. They will be discussed in the next section to enable lexicon scheduling. In Fig. 2, scheduling queue, lexicon queue, result queue and report queue are blocking queues. We use blocking queues for them because they are accessed by multiple threads and this makes sure that the requests and reports are read and written thread safely. Furthermore, when a thread is blocked, it will not consume CPU resource, hence making the system more efficient. Lexicon threads reside in fixed-size thread pool so that we can control the pooling effort.

Algorithm 2 shows the function of a lexicon thread. Because all lexicons update their status simultaneously, it is easier for a lexicon to obtain its neighbors most recent status and the status change of one lexicon can propagate through the network faster than the sequential implementation. In the parallel implementation each lexicon updates its excitation level based on neighbor information and prune inactive symbols asynchronously in a distributed manner.

C. Lexicon scheduling for intermittent pruning

The lexicon thread in Algorithm 2 picks a lexicon from lexicon queue and keeps on processing it until there is only one active symbol. Its effectiveness is based on an assumption that all other lexicons are simultaneously being processed, and the thread has the most up-to-date information on its neighbor status. For many applications, the number of lexicons

Algorithm 2: Lexicon thread confabulation recall algorithm

```
Data: Primary lexicon scheduling queue LexiconQ
Result: Status report queue RepQ
while True do
    l_u ←dequeue(LexiconQ)
    N ← |A_{l_u}|
    while N > 1 do
        for each symbol s ∈ A_{l_u} do
            calculate el(s)
        end
        sort(A_{l_u}) based el(s), ∀s ∈ A_{l_u}
        A_{l_u} ← the first N symbols in A_{l_u}
        for each symbol s ∈ A_{l_u} do
            calculate I(s)
        end
        N ← N − 1
    end
    A*_{l_u} ← A_{l_u}
    ReqQ ← enqueue(l_u)
end
```

is greater than the number of active lexicon threads that can run simultaneously on a processor. If the status of the lexicons that are currently being computed depends on the status of lexicons that have not been processed, then all the calculation and pruning are based on stale information. Again, due to the recurrent knowledge link connections, lexicons have mutual dependencies, and we are not able to find an evaluation order for the lexicons to satisfy all precedence constraints.

Instead of oversubscribing the hardware and using a very large pool of lexicon threads to accommodate all lexicons, and requesting OS to schedule those threads, we limit ourselves to a small thread pool and create our own lexicon scheduling algorithm to share each thread among multiple lexicons. The main idea is to process a lexicon until it reaches a state that its status is informational to its neighbors, then yield the computation resource (i.e. the lexicon thread) to another lexicon. Whether a lexicon status is informational is measured by the normalized difference between the highest and second highest excitation level of the lexicon, and we refer it as the *confidence level (cl)*. It is calculated as the following:

$$cl = min(1, \frac{|el(0) - el(1)|}{max(el(0), el(1))}) \quad (3)$$

where $el(0)$ and $el(1)$ are the highest and second highest excitation level in that lexicon.

As shown in Fig. 2, a local non-blocking queue is attached to each lexicon thread to process lexicons in a round-robin way. Lexicon thread keeps popping up one lexicon from its local queue to run confabulation algorithm. If current lexicon's confidence level exceeds the average *cl* among all lexicons in this recall, then the lexicon will yield the thread and be put back into the local queue for future access, and a new lexicon from the local queue will be fetched for processing. When all lexicons in the local queue have been processed, the thread will move a new lexicon from the lexicon queue to its local queue and repeat the above procedures.

The revised parallel implementation processes each lexicon in an intermittent manner; therefore, we refer it as lexicon level parallel with *intermittent-pruning*. Intuitively if lexicons are pruned too slowly, as they were in the sequential implementation, then there are too many active symbols in the network and they will introduce noise to the recall. On the other hand, if lexicons are pruned too fast, as they were in the simple lexicon level parallel implementation, then there is not enough time for the status change to propagate through the network. The intermittent-pruning finds a balance between these two. It maintains a balanced progress among all lexicon evaluations and by putting a lexicon back to the local queue it gives some time for the lexicons new status to propagate through the network before working on it again.

Fig. 3. Sentence lexicon model example

Interestingly, intermittent pruning also helps to improve the performance, as we will show in the experimental results. This is probably because it reduces the memory contention. Since all running lexicon threads are using shared data, cache coherence [11] needs to be maintained. It takes a lot of time updating the L1/L2 Cache for each CPU core when the excitation levels are read and written by different threads running on different cores simultaneously. Intermittent pruning reduces the frequency that a lexicon is updated, therefore, it reduces the chance of memory access contentions.

IV. Experimental Results

We take sentence reconstruction as an example to evaluate the proposed parallel implementation of associative inference. We implemented our proposed algorithm as a standalone module in ITRS [4], [7], [8]. It receives ambiguous word candidates from noisy scan input, then run confabulation recall to resolve the ambiguity and select appropriate words to reconstruct a sentence. In sentence confabulation model, we assume that the maximum length of a sentence is 20 words and sentences with more than 20 words will be truncated. As Fig. 3 shows, we have three layers of lexicons: words, adjacent word pairs, and *Part-of-speech(POS)* tags [5], [12], [13]. Lexicon 0 to 19 correspond to single English word at location 0 to 19 in a sentence. Lexicons 20 to 38 correspond to 19 word pairs combining word from lexicon 0 to 19 and its right adjacent neighbor. Lexicon 39 to 58 correspond to the POS tag in accordance with each word. Each lexicon stores a tremendous number of symbols (words, word pairs or tags) that appears in the corresponding location. We define intra-level KLs as KLs from one lexicon to another in the same lexicon layer while inter-level KLs are KLs from one lexicon to another lexicon in a different lexicon layer. KLs are created between two inter-level or intra-level lexicons that are less than N-neighborhood (empirically $N = 5$) far away from each other and shared among lexicon pairs that have the same relative position in a sentence.

We took three pages of occluded scanned documents, which are not included in the training corpus of sentence confabulation model. The documents consist of 315 sentences, totally

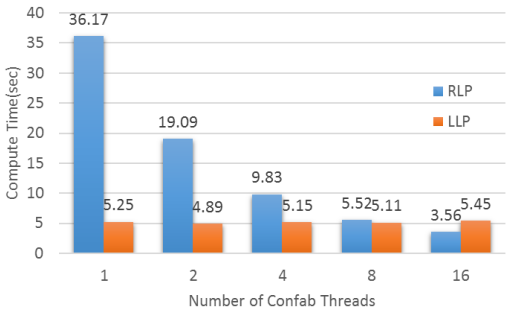

Fig. 4. Compute time for RLP and LLP w/o intermittent pruning

2241 words scanned from a printed paper. About 10% of the words are fully or partially occluded. The noise causes great ambiguity in the input of sentence confabulation. In average there are 5 candidate words at each word location of a sentence, and we need to resolve the ambiguity to recover the sentences.

We measure the compute time as the duration from the end of system initialization to the time when the last sentence has been confabulated. The recall accuracy is measured by sentence accuracy, which specifies the percentage of sentences that are reconstructed exactly same as the original sentence. The tests are implemented on a Linux-2.6.32 based machine with two 4-core CPUs (Intel® Xeon® W5580@3.20GHz with 48GB RAM). The machine has totally 8 cores and 16 logical processors (16 simultaneous threads).

The first group of tests compares the *request level parallelism (RLP)* to *lexicon level parallelism (LLP)*. No intermittent pruning is enabled. For the parallel design, we set lexicon thread pool size to 5 and vary the number of confabulation threads from 1 to 16. When there is only 1 confabulation thread, the RLP implementation reduces to serial implementation. Fig. 4 and Fig. 5 show the compute time and recall accuracy of these two implementations. We can observe that as the number of confab thread increases, the compute time of RLP decrease linearly, and the sentence accuracy of RLP is consistently 69.21% for all five configurations. This is because each confab thread in RLP processes independent recall requests serially. Increasing the number of threads will not affect how the recall function is evaluated. Meanwhile, the compute time of LLP is relatively independent of the number of confabulation threads, because its computation resources are the lexicon threads, whose size is constant in this experiment. Furthermore, because LLP no longer has the overhead of convergence check and it prunes inactive symbols asynchronously, running with 5 lexicon threads in LLP is faster than running with 8 confab threads in RLP. It is interesting to see that using LLP the recall accuracies are also improved visibly. Because parallel confabulation introduces randomness of computation and overcomes the error due to the fixed execution order.

In the second group of tests, we compare lexicon level parallel implementation without intermittent pruning (LLP w/o Itm) with parallel implementation with intermittent pruning (LLP w. Itm). Again we set lexicon thread pool size to 5 for both of them. Fig. 6 shows that the compute time of both implementations are independent to the number of confabulation threads. However, LLP with intermittent pruning

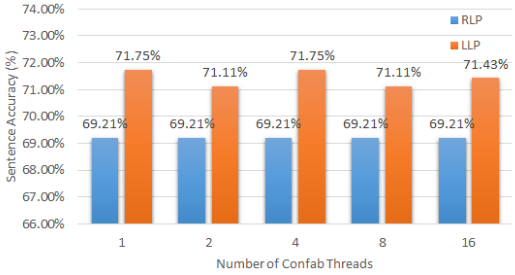

Fig. 5. Sentency accuracy for RLP and LLP w/o intermittent pruning

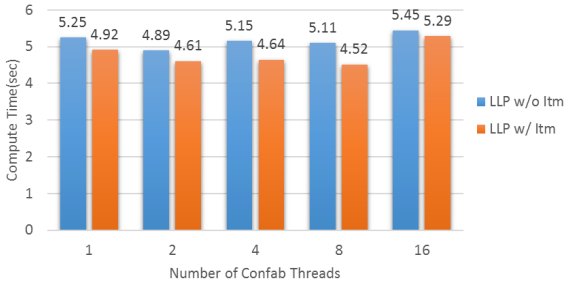

Fig. 6. Compute time for LLP w/o and w/ intermittent pruning

is 7.2% faster than LLP without intermittent pruning. As we explained before, we attribute this phenomenon to less memory contention. Pausing will spread the processing of a lexicon over longer period of time, and hence reduce the chance of memory contention due to simultaneous read and write by different threads, and alleviate the cache coherence overhead.

As for the accuracies, with intermittent-pruning, sentence accuracy improved by about 1%. In the second group compared to the other group. As Fig. 7 shows, the sentence accuracy ranges from 72.38% to 73.02%, which is higher than the other groups accuracies ranging from 71.11% to 71.75%. This is because with intermittent pruning, more lexicons can be loaded to lexicon threads; therefore, it maintains a more balanced progress of status update among different lexicons. Furthermore, a lexicon is pruned through multiple pruning runs instead of one processing. This gives some time for changes to propagate through the network.

In the third group of tests, we compare performance and accuracy of different configurations of LLP with intermittent pruning. The sizes of lexicon pool are set to 5, 10, and 20. We can see that in Fig. 8, given the same number of confab thread, increasing number of lexicon threads improves performance. Moreover, more lexicon threads not only speed up computation, but also increase the accuracies as shown

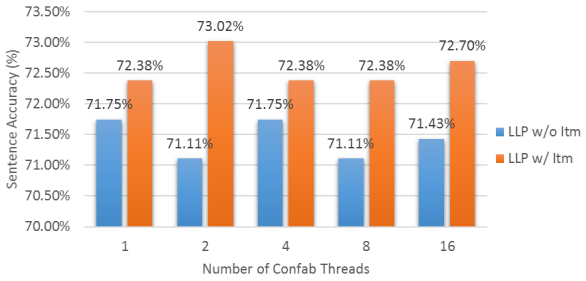

Fig. 7. Sentency accuracy for LLP w/o and w/ intermittent pruning

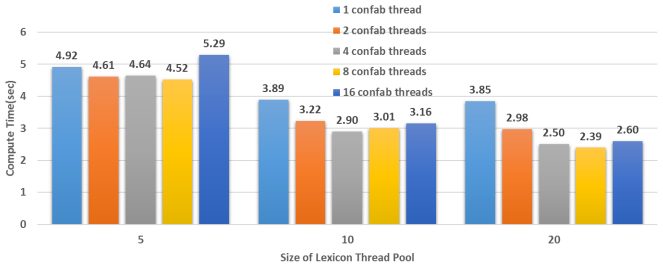

Fig. 8. Compute time for LLP w/ intermittent pruning for different lexicon thread pool sizes

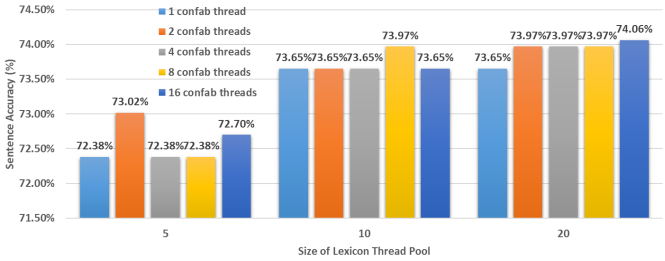

Fig. 9. Sentence accuracy for LLP w/ intermittent pruning with different lexicon thread pool sizes

in Fig. 9. In general, higher parallelism introduces more randomness in processing order and more vividly resemble a real biological neural network. However, the improvement is not significant when we increase lexicon pool size from 10 to 20. This is probably because in average each lexicon is connecting to 10 neighbors (N = 5). We also found that increasing the number confabulation thread does not have noticeable impact on recall accuracy. It does not help to reduce computing time either when the lexicon pool is small. However, when the lexicon pool size increases to 20 threads, increasing the number of confabulation thread can give up to 60% reduction in computing time. This is because we need more confabulation threads to generate inputs and analyze outputs for the large number lexicon threads.

In the last group of tests, to investigate the impact of over-subscription, we run LLP with intermittent pruning on another Linux-2.6.32 based machine with two 8-core CPUs (Intel® Xeon® E5-2690@2.90GHz with 192GB RAM). The machine has totally 16 cores and 32 logical processors (32 simultaneous threads). We compare LLP-Itm running on the 16-core CPU with that running on the 8-core CPU. We set lexicon thread pool size as 20 so that we will oversubscribe the 8-core CPU. As shown in Fig. 10, it is not surprising to see that 16-core CPU is faster than 8-core CPU. What is interesting is that the system on 16-core CPU benefits more from increasing the number of confab threads. When the confabulation thread increases from 1 to 2, it brings approximately 30% compute time reduction in both systems. When we increase the number of confabulation thread to 8, it leads to 60% improvement in the 8-core system but more than 100% improvement in the 16-core system. It receives more than 100% reduction in compute time. This is because not all 20 lexicon threads are active in the 8-core system, and its throughput saturates. However, the accuracy of both systems are very close. This is because oversubscription randomly schedules the lexicon thread to be processed, the randomness of the execution order remains.

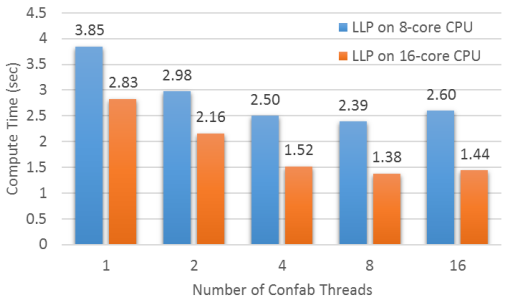

Fig. 10. Compute time for LLP w/ intermittent pruning on 8-core CPU machine and 16-core CPU machine with 20 lexicon threads

V. Conclusion

We proposed a parallel sentence confabulation framework inspired by concurrent association phase of human cognitive processing. The proposed framework exploits multi-threading to build a parallel structure to process lexicons in sentences. We optimized the proposed framework by using intermittent pruning, to overcome the compute speed overhead due to cache coherence and this also improves the accuracy performance of the framework. Compared to request level parallelization, the proposed finer grained parallelization reduces the recall time by up to 93.4%, and increase the sentence accuracy by 5%.

References

[1] D. Bennett and C. Hill, *Sensory integration and the unity of consciousness*. MIT Press, 2014.

[2] G. Kachergis, C. Yu, and R. M. Shiffrin, "An associative model of adaptive inference for learning word–referent mappings," *Psychonomic bulletin & review*, vol. 19, no. 2, pp. 317–324, 2012.

[3] Q. Qiu, Q. Wu, D. J. Burns, M. J. Moore, R. E. Pino, M. Bishop, and R. W. Linderman, "Confabulation based sentence completion for machine reading," in *Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), 2011 IEEE Symposium on*. IEEE, 2011, pp. 1–8.

[4] Q. Qiu, Q. Wu, M. Bishop, R. E. Pino, and R. W. Linderman, "A parallel neuromorphic text recognition system and its implementation on a heterogeneous high-performance computing cluster," *Computers, IEEE Transactions on*, vol. 62, no. 5, pp. 886–899, 2013.

[5] F. Yang, Q. Qiu, M. Bishop, and Q. Wu, "Tag-assisted sentence confabulation for intelligent text recognition," in *Computational Intelligence for Security and Defence Applications (CISDA), 2012 IEEE Symposium on*. IEEE, 2012, pp. 1–7.

[6] Z. Li and Q. Qiu, "Completion and parsing chinese sentences using cogent confabulation," in *Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), 2014 IEEE Symposium on*. IEEE, 2014, pp. 31–38.

[7] Q. Qiu, Z. Li, K. Ahmed, H. H. Li, and M. Hu, "Neuromorphic acceleration for context aware text image recognition," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2014, pp. 1–6.

[8] Q. Qiu, Z. Li, K. Ahmed, W. Liu, S. F. Habib, H. H. Li, and M. Hu, "A neuromorphic architecture for context aware text image recognition," *Journal of Signal Processing Systems*, pp. 1–15, 2015.

[9] R. Hecht-Nielsen, *Confabulation theory: the mechanism of thought*. Springer Heidelberg, 2007.

[10] R. Battiti, "Using mutual information for selecting features in supervised neural net learning," *Neural Networks, IEEE Transactions on*, vol. 5, no. 4, pp. 537–550, 1994.

[11] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[12] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*. Association for Computational Linguistics, 2000, pp. 63–70.

[13] K. Toutanova, D. Klein, C. Manning, W. Morgan, A. Rafferty, M. Galley, and J. Bauer, "Stanford log-linear part-of-speech tagger," 2000.