CCRank: Parallel Learning to Rank with Cooperative Coevolution

Shuaiqiang Wang[1,2*] and **Byron J. Gao**[2] and **Ke Wang**[3] and **Hady W. Lauw**[4]

[1] Shandong University of Finance, 40 Shungeng Road, Jinan, China 250014
[2] Texas State University-San Marcos, 601 University Drive, San Marcos, TX USA 78666
[3] Simon Fraser University, 8888 University Drive, Burnaby, BC, Canada V5A 1S6
[4] Institute for Infocomm Research, 1 Fusionopolis Way, Singapore 138632

Abstract

We propose CCRank, the first parallel algorithm for learning to rank, targeting simultaneous improvement in learning accuracy and efficiency. CCRank is based on cooperative coevolution (CC), a divide-and-conquer framework that has demonstrated high promise in function optimization for problems with large search space and complex structures. Moreover, CC naturally allows parallelization of sub-solutions to the decomposed sub-problems, which can substantially boost learning efficiency. With CCRank, we investigate parallel CC in the context of learning to rank. Extensive experiments on benchmarks in comparison with the state-of-the-art algorithms show that CCRank gains in both accuracy and efficiency.

1 Introduction

Ranking schemes are critical for information retrieval (IR) systems and web search engines. Traditional ranking methods include the Boolean model, vector space model, probabilistic model and language model. Recently, learning to rank has received increasing attention (Joachims 2002; Freund et al. 2003; Cao et al. 2007; Xu and Li 2007; Hoi and Jin 2008; Cao et al. 2010). Given training data, a set of queries each associated with a list of search results labeled by relevance degree, learning to rank returns a ranking function that can be used to order search results for future queries.

With learning accuracy being the primary concern, learning efficiency can be a crucial issue (Cao et al. 2007). Due to diversity of queries and documents, learning to rank involves larger and larger training data with many features. For example, the CSearch dataset used in ListNet (Cao et al. 2007) contains ∼25 million query-document pairs with 600 features. Recently, utilization of click-through data (Joachims 2002) bypasses manual labeling and enables collection of unlimited training data. In addition, due to the rapid growth of the Web, ranking functions need to be re-learned repeatedly. Therefore, it emerged to be an important research prob-lem to achieve high efficiency through parallelization while maintaining accuracy.

In light of this, we propose CCRank, a parallel evolutionary algorithm based on cooperative coevolution (CC), targeting simultaneous improvement in learning accuracy and efficiency. Moreover, CCRank can directly optimize non-derivable evaluation measures and yield non-linear ranking functions, leading to further improvement in accuracy (Liu 2009).

Evolutionary algorithms (EAs) are derived from Darwinian evolutionary principles and widely applied in computationally difficult optimization and classification problems. An EA algorithm maintains a population of individuals (solutions) that evolve from generation to generation. Each individual is associated with a fitness score. In standard EAs, fitness scores are static (do not vary over time) and absolute (independent of other individuals).

Cooperative coevolution (CC) is a special type of EAs, where the fitness of an individual is based on how well it cooperates with other interacting individuals. CC algorithms are advantageous in solving problems with exceptionally large search space and complex structures (Wiegand 2004). They have been successfully applied to a variety of domains including function optimization (Potter and Jong 1994), manufacturing scheduling (Phil Husbands 1991), etc. CC algorithms follow a divide-and-conquer strategy, decomposing a complex problem into sub-problems and combining sub-solutions (individuals) in the end to form the final complete solution.

In CC algorithms, EAs evolve separately. Thus, the evolving process can be naturally parallelized, allowing significant improvement in learning efficiency. Unfortunately, this privilege has not been explored previously for learning to rank. In this paper, we investigate parallel CC in the context of learning to rank, targeting simultaneous improvement in accuracy and efficiency.

Contributions. Our main contributions are as follows. (1) We investigate parallel cooperative coevolutionary algorithms in the context of learning to rank. (2) We propose CCRank, the first parallel algorithm for learning to rank, targeting simultaneous improvement in learning accuracy and efficiency. (3) We present extensive experiments on benchmarks, demonstrating the promise of CCRank in comparison with the state-of-the-art algorithms.

2 Related Work

Learning to rank. Learning to rank has received increasing attention recently from both machine learning and IR communities. Now we review several representative algorithms. RankSVM (Joachims 2002; 2006) is based on support vector machines using click-through data for training. It minimizes the number of discordant pairs (by the final ranking function) for margin maximization. RankBoost (Freund et al. 2003) adopts a Boosting approach for learning to rank. It minimizes the weighted number of disordered pairs. AdaRank (Xu and Li 2007) is also a Boosting approach, but it minimizes a loss function directly defined on the performance measures. ListNet (Cao et al. 2007) introduces a probabilistic list-wise loss function, and uses neural network and gradient descent to train a list prediction model.

Other algorithms are surveyed in (Liu 2009). Differing from all of these algorithms, CCRank is a parallel algorithm, targeting improvement in accuracy and efficiency.

Parallel machine learning. Many sophisticated machine learning algorithms cannot process large data sets. Parallelization is an effective way of achieving speed-up. IBM Parallel Machine Learning Toolbox (PML)[1] contains the parallel version of many commonly-used machine learning algorithms (e.g., SVM), and includes an API for incorporating additional algorithms. The toolbox can work on various types of architecture, e.g., multicore machines. By distributing the required computation to computing nodes in a parallel fashion, training can be expedited by several orders of magnitude.

While not extensively studied, with the advances of multicore technology, parallel machine learning is emerging as an active research discipline. For example, Proximal SVM (Collobert, Bengio, and Bengio 2004) and Cascade SVM (Graf et al. 2004) are parallel SVMs. (Chu et al. 2006) attempts to provide a simple and unified framework for parallel machine learning. However, to our best knowledge, no parallel machine learning algorithms have been applied to the learning to rank problem.

Cooperative coevolution. Cooperative coevolution (Potter and Jong 1994) is a divide-and-conquer coevolutionary architecture for solving problems with exceptionally large search space and complex structures. The principles of CC will be introduced shortly in Section 3.

Early CC-based algorithms, e.g., JACC-G (Yang et al. 2009), CCEA (Potter and Jong 1994), CCPSO (Li and Yao 2009), aim at function optimization. CCRank is the first algorithm adapting CC to learning to rank.

3 The Learning to Rank Problem

Let $\mathcal{D}$ be a collection of documents, each represented by a vector of feature values. In an information retrieval system, for a query q, a list of documents from $\mathcal{D}$ are returned as search results, where the documents are ranked according to their relevance to q.

For a given query q, the ground truth relevance of documents with respect to q (judged by human experts) is defined

[1]http://www.alphaworks.ibm.com/tech/pml

as a function $rel : \mathcal{D} \to \mathbb{N}$, where $\mathbb{N}$ is the natural number set indicating different relevance levels. In some cases, rel is a binary function, mapping a document to either 0 (irrelevant) or 1 (relevant). In our experiments, we considered 3 relevance levels of 0 (irrelevant), 1 (partially relevant), and 2 (relevant).

Let $f : \mathcal{D} \to \mathbb{R}$ be a ranking function assigning real number relevance scores to documents, where $\mathbb{R}$ denotes the real number set. The goodness of ranking functions can be evaluated by a given measure s, such as precision at n ($P@n$), mean average precision (MAP), and normalized discount cumulative gain ($NDCG@n$).

Definition 1 *Given a training data set $\mathcal{T}$, given an evaluation measure s, the problem of learning to rank is to learn a ranking function f from $\mathcal{T}$ such that $s(f)$ is maximized.*

4 The CCRank Algorithm

4.1 Overview of CCRank

CCRank adapts parallel cooperative coevolution (CC) to the learning to rank problem. It learns a ranking function from training data in two phases.

CCRank starts with the *problem decomposition phase* (detailed in Section 4.3). Firstly L initial solutions, represented by trees, are generated randomly from the full feature space. Then, each tree is decomposed into N sub-trees, resulting in N populations each with L individuals.

The *evolution phase* (detailed in Section 4.4) starts after problem decomposition. It is an iterative process, where N populations co-evolve in parallel from generation to generation. Each population maintains a collection of individuals and a *winner*, which is the best individual in the population with the biggest fitness value.

At the end of each generation, the parallel execution is suspended and a complete solution f is produced by a combination operation, which assembles N winners evolved in N populations. Then f is collected into a solution pool as a candidate ranking function.

After the evolution process ends, validation data are used to select the best solution among all candidates as the final ranking function to return.

4.2 Solution Representation

CC-based algorithms for optimization problems use vectors to represent solutions. For the learning to rank problem, the ranking function to be learned, f, can be nonlinear. In CCRank, we use tree structures to represent solutions. Accordingly, individuals are represented as subtrees. Trees not only have sufficient expressive power to represent non-linear functions (de Almeida et al. 2007; Fan, Gordon, and Pathak 2004), they also have the advantage of allowing easy parsing, implementation and interpretation.

In particular, for each tree, the internal nodes contain basic function operators of $+$, $-$ and $*$. The leaf nodes contain features and constants. Constants serve as coefficients of features in f. In CCRank, 19 constants are used, which are 0.1, 0.2, ..., 0.9, 1, 2, ..., 10.
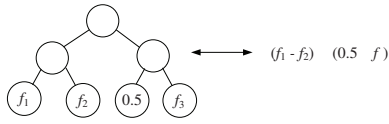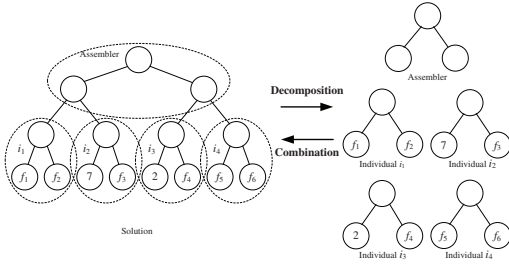
Figure 1: Tree representation

Figure 2: Decomposition and combination

Figure 3: Evolution

A tree can be parsed into a function. Figure 1 shows the tree representation for an example ranking function $(f_1 - f_2) + (0.5 * f_3)$.

The depth d of a tree representing a complete solution is determined by the total number of features n_F and the total number of constants n_C. An empirical design (Fan, Gordon, and Pathak 2004; Wang, Ma, and Liu 2009) has been that the tree should be deep enough so that the number of leaf nodes is bigger than $n_F + n_C$, i.e., $d = \lceil \log_2(n_F + n_C) \rceil + 1$. For example, let $n_F = 46$ and $n_C = 19$, then $d = \lceil \log_2(46 + 19) \rceil + 1 = 7 + 1 = 8$. A tree of depth 8 has $2^{8-1} = 128 \geq (46 + 19)$ leaf nodes.

4.3 Decomposition Phase

CC-based optimization algorithms divide the feature space into subspaces of features, each corresponding to a sub-problem. This decomposition is appropriate because their search space is a Cartesian product of features. In CCRank, the search space is non-linear. A similar decomposition would lead to significant loss of information and compromised search space.

In CCRank, initially L solutions, represented by trees of depth d, are generated randomly from the full feature space. Then, each tree is decomposed into N sub-trees, resulting in N populations, each with L individuals. Each population will be assigned an EA to evolve. Figure 2 shows the decomposition of a single tree (left-hand side) into $N = 4$ individuals (right-hand side).

The depth of sub-trees (individuals) d_I upper-bounds the feature space of individuals. This parameter is used in CCRank whenever individuals are generated. The depth of assembler (to be explained in Section 4.4) is determined by $d_A = \lfloor \log_2 n_P \rfloor$ where n_P is the number of processors used in the parallel evolution process. Thus, $d_I = d - d_A$, which implies that the size of the feature space of individuals is upper-bounded by 2^{d_I-1}. For example, suppose the depth of trees (solutions) $d = 8$. The calculation of d was introduced in Section 4.2. Let the number of processors $n_P = 8$. Then, the depth of assembler $d_A = \lfloor \log_2 8 \rfloor = 3$, and the depth of individuals $d_I = 8 - 3 = 5$.
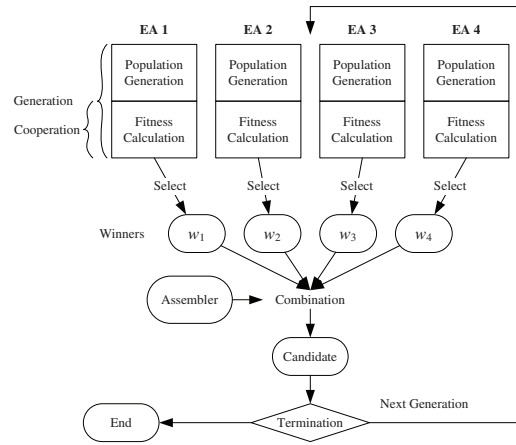
4.4 Evolution Phase

Due to the differences in search spaces, linear vs. non-linear, the evolution process differs between optimization algorithms and CCRank. In the former, individuals evolve within the same predetermined subspace. In the latter, an *open* approach is adopted in the sense that any feature from the full feature space can be selected into the subspace.

Evolution in CCRank executes iteratively, and the number of iterations is predetermined by a given parameter. Each iteration contains a generation of evolving populations. N populations evolve in parallel, each maintaining a collection of individuals and a *winner*, which is the best individual in the population with the biggest fitness value.

Figure 3 illustrates the iterative evolution process for 4 populations in EA1, EA2, EA3 and EA4 respectively. Within each iteration, there is a generation as shown at the top. Within each generation, in the beginning, individuals of each population for the current generation are generated based on individuals from the previous generation. Then, fitness values of individuals are calculated in a cooperative manner, as to be detailed shortly.

After the generation, parallel execution is suspended temporarily and the candidate solution for the current generation will be generated. First, the winner for each population is updated based on the calculated fitness. Then, all the winners are combined, with the help of the assembler, into a candidate solution. If the preset maximum number of iterations is not met, a new iteration will start and the 4 populations will continue to evolve in a new generation in parallel.

Combination Operation. The combination process assemblies individuals into a complete solution. For optimization algorithms, combination is straightforward since individuals are vectors. Besides, combination is done in the very end of all generations. In CCRank, individuals are sub-trees and we need to assemble them properly to form a complete solution. In addition, since learning to rank is a classification problem that requires validation of candidate classifiers, each being a solution, we need to collect a pool of diverse candidates produced by different generations. Thus, in CCRank combination is done after each generation during the evolution
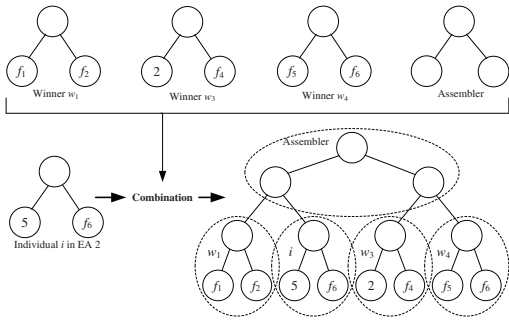
Figure 4: Fitness calculation

process. Precisely, combination occurs in two cases: fitness calculation (right *before* the end of the generation, as shown in Figure 4) and candidate generation (right *after* the end of the generation as shown in Figure 3).

Combination is the inverse process of decomposition. In Figure 2, 4 individuals $i_1 \sim i_4$ (right-hand side) are combined into a complete solution (left-hand side) with the help of the assembler. *Assemblers* are used to assemble individuals into complete solutions, where they form "crowns" of solution trees. Thus, all of their nodes are internal nodes of solution trees and contain function operators only. In particular, we use $+$ so as to generate simple ranking functions.

Fitness calculation. Under the CC framework, fitness of individuals is based on how well they cooperate with other populations. Figure 4 illustrates the fitness calculation for individual i in EA2, one of the 4 EAs in Figure 3. First, individual i and winners w_1, w_3 and w_4 selected in EA1, EA3 and EA4 from the *previous* generation, with the help of the assembler, are combined into a solution (right-hand side). Then, the evaluation measure, e.g., MAP, for the combined solution is calculated using training data, and the resulting score is assigned to individual i as its fitness.

The fitness values of other individuals are calculated in the same manner. It seems that fitness calculation requires cooperation involving mutual dependency, which would make parallel execution of EAs infeasible. However, note that the cooperation is between two *different* generations. As shown in Figure 4, individual i is from the *current* generation, winners w_1, w_3 and w_4 are the best individuals from the *previous* generation. Thus, there is no mutual dependency and all EAs can perform fitness calculation in parallel.

Choice of EA. In CCRank we choose Immune Programming (IP) (Musilek et al. 2006) as our evolutionary algorithm. This is because comparing to Genetic Programming (GP), IP can evolve a good solution in fewer generations, especially when the population size is small (Musilek et al. 2006). Moreover, IP has been successfully applied to the learning to rank problem (Wang, Ma, and Liu 2009).

4.5 The Algorithm

We have explained the main procedures of CCRank, now we summarize them and present the pseudocode of CCRank in Algorithm 1.

In the pseudocode, $\mathcal{P}_i^{(g)}$ denotes population i in the gth generation. w_i denotes the winner of population i. A de-

---

Algorithm 1: CCRank

Input : Training set $\mathcal{T}$, validation set $\mathcal{V}$, maximum number of generations G, number of populations N

Output: Ranking function f

1 Initialize ()   // decomposition
2 **for** $g \leftarrow 1$ **to** G **do**
3   **for** $i \leftarrow 1$ **to** N **do**   // parallel evolution
4        $\mathcal{P}_i^{(g)} \leftarrow$ Evolve $(\mathcal{P}_i^{(g-1)})$ // CC
5   Update $(w_i, \mathcal{P}_i^g)$
6     $f^{(g)} \leftarrow$ Combine $(\{w_i\}_{i=1}^N, A)$// combination
7     $\mathcal{C} \leftarrow \mathcal{C} \cup \{f^{(g)}\}$
8 $s_{\mathcal{T}}(C) \leftarrow$ Evaluate $(\mathcal{C}, \mathcal{T})$
9 $s_{\mathcal{V}}(C) \leftarrow$ Evaluate $(\mathcal{C}, \mathcal{V})$
10 $f \leftarrow$ Select $(s_{\mathcal{T}}(C), s_{\mathcal{V}}(C))$

---

notes an assembler. $\mathcal{C}$ denotes the candidate set. s denotes the fitness function of individuals.

Line 1 performs initialization, including random generation of N populations, evaluation of each individual, and selection of initial winners.

Lines 2-7 show the whole evolution process from generation to generation. Specifically, lines 3-5 show one generation of the evolution process for all populations that execute in parallel. In particular, line 4 evolves individuals based on the previous generation, and line 5 updates the winner for each EA. Lines 6-7 generate candidate solutions.

Lines 8-9 calculate the performance measures $s_{\mathcal{T}(\mathcal{C})}$ and $s_{\mathcal{V}(\mathcal{C})}$ for the candidates using the training set $\mathcal{T}$ and validation set $\mathcal{V}$, based on which line 10 selects the ranking function f. The following formula (Wang, Ma, and Liu 2009) is used in the selection:

$$\arg\max_i \left((\alpha \times s_{\mathcal{T}}(f_i) + \beta \times s_{\mathcal{V}}(f_i)) - \gamma \times \sigma_i \right),$$

where γ is a constant, σ_i is the standard deviation of $s_{\mathcal{T}}(f_i)$ and $s_{\mathcal{V}}(f_i)$, and the values of α and β are based on the sizes of the training set and validation set, i.e., $\alpha = \frac{|\mathcal{T}|}{|\mathcal{T}|+k\times|\mathcal{V}|}, \beta = \frac{k\times|\mathcal{V}|}{|\mathcal{T}|+k\times|\mathcal{V}|}$.

5 Experiments

We conducted two series of experiments using benchmark datasets to evaluate the accuracy and efficiency performance of CCRank.

5.1 Methodology

Datasets. We used LETOR 4.0, a collection of benchmarks released in 2009 by Microsoft Research Asia (research.microsoft.com/en-us/um/beijing/projects/letor/) because the accuracy of several baseline algorithms were also available online.

LETOR 4.0 uses the Gov2 Web page collection and two query sets from Million Query track of TREC 2007 and TREC 2008, called MQ2007 and MQ2008. There are about

Table 1: Accuracy in MAP

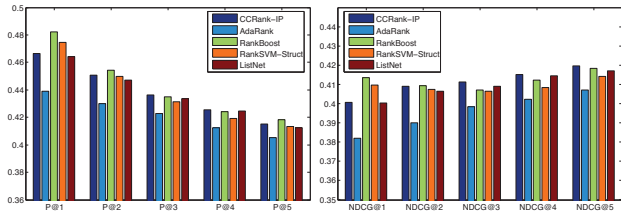| Data	CCRank	AdaRank	RankBoost	RankSVM	ListNet
MQ2007	0.466	0.458	0.466	0.465	0.465
MQ2008	0.482	0.476	0.478	0.470	0.478

Figure 5: $P@n$ and $NDCG@n$ on MQ2007

1,700 queries with 69,623 instances in MQ2007 and about 800 queries with 15,211 instances in MQ2008. Each data set has been partitioned into five parts in order to conduct 5-fold cross validation. For each fold, three parts are used for training, one part for validation, and the remaining part for testing.

Features. We used the features provided by LETOR 4.0. For each document, there are 6 hyperlink features (PageRank value, inlink number, outlink number, number of slashes in URL, length of URL, and number of child pages) and 40 content features consisting of 20 classical features such as document length and term frequency, and 20 high level features such as results of BM25 (Robertson 1997) and LMIR (Zhai and Lafferty 2001) algorithms.

Evaluation measures. Commonly three standard rank-aware accuracy measures are used to evaluate the rank functions generated by learning to rank algorithms: precision at n ($P@n$), mean average precision (MAP), and normalized discount cumulative gain ($NDCG@n$). $P@n$ measures the accuracy within the top n results of the returned ranked list for a query:

$$P@n = \frac{\# \text{ of relevant docs in top } n \text{ results}}{n}.$$

MAP takes the mean of the average precision values over all queries, where the average precision for each query is defined as the average of the $P@n$ values for all relevant documents:

$$average\ precision = \frac{\sum_{n=1}^{N} \left(P@n \times rel(n) \right)}{\# \text{ relevant docs for this query}},$$

where $rel(n)$ is a binary function mapping a document to either 1 (relevant) or 0 (irrelevant).

Note that $P@n$ and MAP can only handle cases with binary judgment, relevant or irrelevant. Recently, a new evaluation measure $NDCG@n$ (Järvelin and Kekäläinen 2002) has been proposed to handle multiple levels of relevance:

$$NDCG@n = Z_n \sum_{j=1}^{n} \begin{cases} 2^{rel(j)} - 1 & \text{if } j = 1 \\ \dfrac{2^{rel(j)} - 1}{log(j)} & \text{if } j > 1 \end{cases},$$
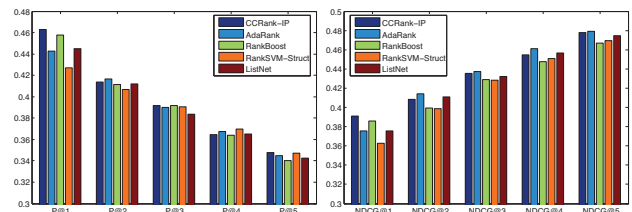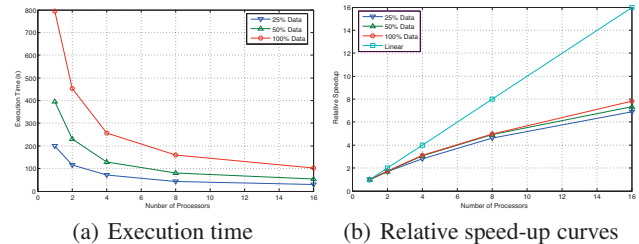

Figure 6: $P@n$ and $NDCG@n$ on MQ2008

(a) Execution time   (b) Relative speed-up curves

Figure 7: Efficiency of CCRank

where $rel(j)$ is the integer rating of the jth document, and the normalization constant Z_n is chosen such that the perfect list gets a $NDCG$ score of 1.

Comparison partners. We compared CCRank with the state-of-the-art learning to rank algorithms AdaRank (Xu and Li 2007), RankBoost (Freund et al. 2003), RankSVM (Joachims 2002) and ListNet (Cao et al. 2007). Their experiment results on datasets MQ2007 and MQ2008 are publicly available (http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4baseline.aspx).

Parameter setting. Our implementation of CCRank used Immune Programming (IP) as the EA algorithm, which has demonstrated advantages in learning to rank (Wang, Ma, and Liu 2009). $N = 8$ EAs are maintained, each containing $L = 70$ individuals that co-evolve up to $G = 30$ generations. The depth of complete solutions is $d = 8$.

5.2 Accuracy

In the first series of experiments, we evaluated the accuracy performance of CCRank, in comparison with AdaRank, RankBoost, RankSVM-struct and ListNet on benchmarks MQ2007 and MQ2008.

Table 1 shows the accuracy comparison under the MAP measure. For MQ2007, CCRank and RankBoost shared the best performance. For MQ2008, CCRank outperformed all other algorithms, gaining 1.13%, 0.901%, 2.60% and 0.901% respectively. Note that the best and worst performances from those comparison partners differ by merely 1.67%.

Figures 5 and 6 show the accuracy comparison under the $P@1{\sim}5$ and $NDCG@1{\sim}5$ measures. The results are consistent with the ones under MAP, showing CCRank is among the best for both measures, comparable to RankBoost on MQ2007 and AdaRank on MQ2008.

5.3 Efficiency

To demonstrate the gain in efficiency by parallel evolution, we extracted 25%, 50%, and 100% portions of MQ2008

and generated 3 datasets, which have 3,803, 7,606, and 15,211 instances respectively. Then we ran CCRank on these datasets varying the number of processors (1, 2, 4, 8, and 16).

Figure 7 shows the execution time and corresponding relative speed-up curves of CCRank. From the results we can see that parallel evolution leads to significant speed-up in CCRank. Comparing to the case of 1 processor, the averaged relative speed-up ratios are 173%, 299%, 486%, and 736% respectively, for the cases of 2, 4, 8, and 16 processors.

Note that ideally with the increase of dataset size, the speed-up curve should approximate the linear line as shown in 7(b). However, CCRank has difficulties in achieving this ideal scale-up for the following reasons. Firstly, as shown in Figure 3, CCRank does not always execute in parallel. After each generation, it must suspend the parallel execution to perform combination in order to produce the candidate solution. Secondly, EAs may spend different amounts of time to evolve for a certain generation, but the combination can start only after all EAs finish. Thus, the time CCRank spends on a generation is equal to the longest time that any EA can possibly spend for the generation.

6   Conclusion

In this paper we proposed CCRank, a parallel learning to rank algorithm based on cooperative coevolution, targeting simultaneous improvement in accuracy and efficiency. While early CC-based algorithms have been successfully applied to complex function optimization problems, we adapted CC to the learning to rank problem. In addition, we also designed mechanisms that allow the cooperatively evolving EAs to execute in a parallel manner. We experimentally compared CCRank with the state-of-the-art algorithms on benchmark datasets, demonstrating the accuracy and efficiency gain of CCRank.

For future work, we plan to extend CCRank in several directions. On direction is to further explore our parallel CC framework by incorporating some recently proposed EA algorithms. Another direction is to investigate cooperative learning to rank. In CC, the cooperating components are EAs. It is possible and interesting to organize other machine learning algorithms, e.g., SVM and Neural Network, to work in a collaborative manner for the learning to rank problem. Last but not least, our current parallel CC framework has demonstrated significant speed-up, but not scale-up. We plan to investigate how to further boost efficiency by taking full advantage of parallelization. For this purpose, more economic and sophisticated cooperation schemes need to be considered.

References

Cao, Z.; Qin, T.; Liu, T.-Y.; Tsai, M.-F.; and Li, H. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proc. of ICML'07*.

Cao, B.; Shen, D.; Wang, K.; and Yang, Q. 2010. Clickthrough log analysis by collaborative ranking. In *Proc. of AAAI'10*.

Chu, C. T.; Kim, S. K.; Lin, Y. A.; Yu, Y.; Bradski, G. R.; Ng, A. Y.; and Olukotun, K. 2006. Map-reduce for machine learning on multicore. In *Proc. of NIPS'06*.

Collobert, R.; Bengio, Y.; and Bengio, S. 2004. A parallel mixture of svms for very large scale problems. In *Proc. of NIPS'04*.

de Almeida, H. M.; Gonçalves, M. A.; Cristo, M.; and Calado, P. 2007. A combined component approach for finding collection-adapted ranking functions based on genetic programming. In *Proc. of SIGIR'07*.

Fan, W.; Gordon, M. D.; and Pathak, P. 2004. Discovery of context-specific ranking functions for effective information retrieval using genetic programming. *IEEE Trans. Knowl. Data Eng.* 16(4):523–527.

Freund, Y.; Iyer, R.; Schapire, R. E.; and Singer, Y. 2003. An efficient boosting algorithm for combining preferences. *J. Mach. Learning Res.* 4(1):933–969.

Graf, H. P.; Cosatto, E.; Bottou, L.; Durdanovic, I.; and Vapnik, V. 2004. Parallel support vector machines: The cascade SVM. In *Proc. of NIPS'04*.

Hoi, S. C. H., and Jin, R. 2008. Semi-supervised ensemble ranking. In *Proc. of AAAI'08*.

Järvelin, K., and Kekäläinen, J. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Sys.* 20(4):422–446.

Joachims, T. 2002. Optimizing search engines using clickthrough data. In *Proc. of KDD'02*.

Joachims, T. 2006. Training linear SVMs in linear time. In *Proc. of KDD'06*.

Li, X., and Yao, X. 2009. Tackling high dimensional nonseparable optimization problems by cooperatively coevolving particle swarms. In *Proc. of CEC'09*.

Liu, T.-Y. 2009. Learning to rank for information retrieval. *Found. Trends Inf. Retr.* 3(3):225–331.

Musilek, P.; Lau, A.; Reformat, M.; and Wyard-Scott, L. 2006. Immune programming. *Inf. Sci.* 176(8):972–1002.

Phil Husbands, F. M. 1991. Simulated co-evolution as the mechanism for emergent planning and scheduling. In *Proc. of GA'91*.

Potter, M. A., and Jong, K. A. D. 1994. A cooperative coevolutionary approach to function optimization. In *Proc. of PPSN'94*.

Robertson, S. E. 1997. Overview of the okapi projects. *J. Doc.* 53(1):3–7.

Wang, S.; Ma, J.; and Liu, J. 2009. Learning to rank using evolutionary computation: Immune programming or genetic programming? In *Proc. of CIKM'09*.

Wiegand, R. P. 2004. *An Analysis of Cooperative Coevolutionary Algorithms*. Ph.D. Dissertation, George Mason University, Fairfax, VA, USA.

Xu, J., and Li, H. 2007. AdaRank: a boosting algorithm for information retrieval. In *Proc. of SIGIR'07*.

Yang, Z.; Zhang, J.; Tang, K.; Yao, X.; and Sanderson, A. C. 2009. An adaptive coevolutionary differential evolution algorithm for large-scale optimization. In *Proc. of CEC'09*.

Zhai, C., and Lafferty, J. 2001. A study of smoothing methods for language models applied to Ad Hoc information retrieval. In *Proc. of SIGIR'01*.