Exploring Scalable Parallelization for Edit Distance-Based Motif Search

Junqiao Qiu and Ali Ebnenasir

Abstract—Motif Searching is an important problem that can reveal crucial information from biological data. Since the general motif searching is NP-hard and the volume of biological data is growing exponentially in recent years, there is a pressing need for developing time and space-efficient algorithms to find motifs. In this paper, we explore scalable parallelization for Edit Distance-Based Motif Search (EMS). We introduce two parallel designs, recursEMS which integrates the existing EMS solver into a parallel recursion tree running in multiple processes, and parEMS that presents a novel thread-based method which avoids the storage of redundant motif candidates. To make the parallel designs practical, we implement *SPEMS*, a Scalability-sensitive Parallel solver for EMS. For any given biological dataset and search instance, SPEMS can provide an EMS parallelization towards the optimal performance, or a sub-optimal performance but being more space efficient. Evaluations on two real-world DNA dataset *TRANSFAC* and *ChIP-seq* show that SPEMS can obtain 10× geometric mean speedup over the state-of-the-art at the expense of no less than 74.7% memory overheads, or provide 2.2× geometric mean speedup with the possibility of consuming less memory, when running on a 48-core machine.

Index Terms—Edit-distance motif search, parallelism

1 Introduction

Biological data mining plays a critical role in a large group of real-world applications ranging from protein function domain detection and function inference to disease diagnosis and treatment optimization [1]. Among various biological data mining problems, a significant one is Edit distance-based Motif Search (EMS), whose objective is to find all common patterns with a certain length appearing in a set of biological sequences (e.g., genes or proteins strings). EMS has applications in identifying characteristic functional units and detecting rare events occurring in biological sequences [2], [3]. While there are numerous methods for solving EMS, there is a pressing need for algorithms and tools that scale well in terms of both time and space when running on modern parallel architectures (e.g., multi-cores CPUs and GPUs). This paper presents a step towards developing parallel EMS solvers that strike a balance between execution time and memory usage.

Most existing approaches simply consider the algorithms of EMS solvers as "naturally parallel," based on the observation that the targeted biological sequences are independent from each other, but they poorly exploit such intrinsic parallelism. For example, Grundy et al. [4] developed a parallel implementation and a web interface for a DNA and protein motif discovery tool called MEME [5] on supercomputers. Qin et al. [6] build up a bioinformatics tool for searching motifs with certain structural and biochemical properties in DNA or protein sequences. Ferretti et al. [7] parallelize structural motif search in proteins on distributed and shared memory systems. Recently, Pal et al. [8] present a compressed tree structure to store candidate motifs, however, this tree cannot easily be used in shared memory-based parallel platforms. Instead, the authors of [8] utilize a simple array to store candidate motifs, which greatly undermines the efficiency of parallelization. This is an issue related to porting the highly optimized sequential EMS solvers into parallel platforms, where there is an urgent need for efficient data structures that enable concurrent data storage and retrieval. Moreover, most existing parallelization methods only have an emphasis on time efficiency, whereas considering both time and space efficiency makes the design of parallel EMS solvers more appealing yet more challenging.

Contributions. In order to address the aforementioned challenges, this paper provides two parallel algorithms for EMS. Specifically, we present:

- a time-efficient parallel algorithm, called recursEMS, that introduces the logic of a recursive EMS algorithm in a parallel fashion. recursEMS outperforms EMS2 [8] and EMS3 [9] methods in terms of time, however, underperforms in terms of memory costs;
- a hash map-based parallel algorithm, called parEMS, for exploring the possibility of achieving both time and space-efficient EMS on shared-memory multicores platforms;
- the proofs of correctness of recursEMS and parEMS, as well as complexity analysis of these parallel algorithms;
- the implementations of a Scalability-sensitive Parallel solver for EMS (SPEMS) which integrates the two proposed parallel EMS algorithms;
- evaluations on two real-world DNA datasets (*TRANSFAC* and *ChIP-seq*) and a synthetic dataset, and hard instances of EMS where we validate the effectiveness of recursEMS and parEMS in terms of both time and space costs; and
- experimental results which show that SPEMS can provide 10× geometric mean speedup over the state-of-the-art at the expense of no less than 74.7% memory overheads, or provide 2.2× geometric mean speedup with the possibility of consuming less memory, when running on a 48-core machine. While recursEMS is the fastest algorithm (to the best of our knowledge), parEMS strikes a balance between time and space efficiency.

The rest of this paper is organized as follows. Terminology used and the formal statement of EMS problem are provided in Section 2. Then Section 3 presents the two proposed parallel EMS algorithms, each followed by their complexity analyses, and Section 4 shows implementations of the two parallel algorithms in SPEMS. Section 5 presents our experimental results. Some related work will be discussed in Section 6. Finally, Section 7 makes a brief conclusion and introduces some future works.

2 Preliminaries

This section first introduces terminology used in the rest of the paper, then formally states the EMS problem and the existing solvers.

2.1 Motif Searching

Two strings x and y in an alphabet Σ are in Levenshtein distance (a.k.a. edit distance) d of each other *iff* (if and only if) x can be obtained from y (and vice versa) by d operations of deleting, replacing or inserting symbols of Σ in any position of y (respectively, x). A string y with length l (an l-mer) is a *d-neighbor* of x *iff* y has an edit distance of at most d from x, thus the length of x is between $l-d$ and $l+d$. The set of all strings y that can be *obtained* from x with an edit distance of at most d is called the d *neighborhood* of x, denoted $N_d(x)$. We also say that y is *obtainable* from x, which reflects the *symmetric* relation.

- *Junqiao Qiu is with the Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong. E-mail: junqiqiu@cityu.edu.hk.*
- *Ali Ebnenasir is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA. E-mail: aebnenas@mtu.edu.*

Manuscript received 22 December 2021; revised 20 July 2022; accepted 11 September 2022. Date of publication 23 September 2022; date of current version 3 April 2023. (Corresponding author: Junqiao Qiu.)
Digital Object Identifier no. 10.1109/TCBB.2022.3208867

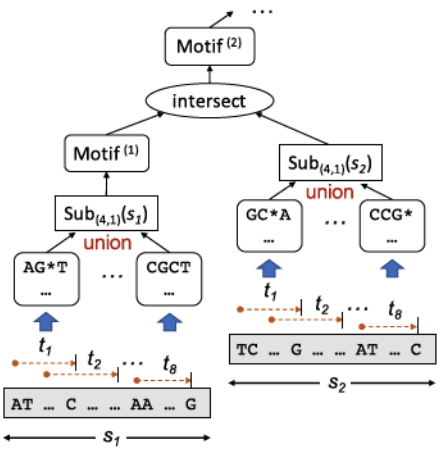


Fig. 1. Illustration of applying the parallel design in [8] to find out all $(4,1)$-motifs in S with 8 sequences, assuming there are 8 available processing units.

Lemma 1 (Symmetry). *y is in edit distance d of x iff x is in edit distance d of y.*

Previous work [10], [11] proves Lemma 1 by showing that the basic operations of insertion, deletion and substitution are reversible. Let $Sub(s)$ denote the set of all non-null substrings of a sequence s. We follow [8] in defining $Sub_{(l,d)}(s)$ as the set of strings of length l within the edit distance d of any substring $x \in Sub(s)$ of length $l-d \leq k \leq l+d$. In other words, the set $Sub_{(l,d)}(s)$ is the union of all $N_d(x)$ of length l for any k-mer x, where $l-d \leq k \leq l+d$.

Definition 1 ((l,d)-motif). *x is an (l,d)-*motif *(with length l and distance d) of two sequences s_1 and s_2 iff x appears in s_1 and s_2 with an edit distance d. That is, x belongs to the intersection of $Sub_{(l,d)}(s_1)$ and $Sub_{(l,d)}(s_2)$.*

2.2 Problem Statement

This section states the problem of Edit distance-based Motif Search (EMS). The input to the problem is a set S of n independent sequences while each sequence $s_i \in S$ has a length $m_i > 0$. While in general sequences of S can have different lengths, for simplicity of our analysis we make the assumption that they have the same length; i.e., $m_i = m$ for $0 < i \leq n$. The objective is to find out all strings of length l that appear in each s_i with an edit distance no more than d.

Problem 1 (EMS). *For a given set S of $n > 0$ sequences, and integers l and d, find all (l,d)-motifs of S.*

By generalizing Definition 1, we know x is an (l,d)-motif of a set S of n strings of length m *iff* $x \in (Sub_{(l,d)}(s_1) \cap Sub_{(l,d)}(s_2) \cdots \cap Sub_{(l,d)}(s_n))$. This *intersection*-oriented definition builds up the foundation for parallel EMS solvers.

State-of-the-Art. There has been a significant amount of research efforts on solving the EMS problem (as discussed in Section 6). Recent works focus on developing efficient neighbourhood generation [8], [9]. Their key idea is to generate and store compact d-neighbors (which are the candidate motifs) with wildcard characters. These works [8], [9] claim that EMS solvers can be *nicely parallelized*. Specifically, in [8], each sequence is divided into chunks and then available threads will process over these chunks in parallel (with the existing EMS solver). Fig. 1 illustrates an example. Eight processing units are available when all $(4,1)$-motifs in a set S of 8 sequences should be found. This parallel design first applies all processing units on the first sequence s_1, and makes processing units execute the existing EMS solver over the divided chunks. When this parallel execution is finished, we can construct the *union* of all generated d-neighbours and thus get $Sub_{(4,1)}(s_1)$. The processing units then move to the second sequence and construct $Sub_{(4,1)}(s_2)$. By generating the *intersection* of $Sub_{(4,1)}(s_1)$ and $Sub_{(4,1)}(s_2)$, all $(4,1)$-motifs in the first two sequences are collected. Following the same strategy, the original EMS problem can be solved.

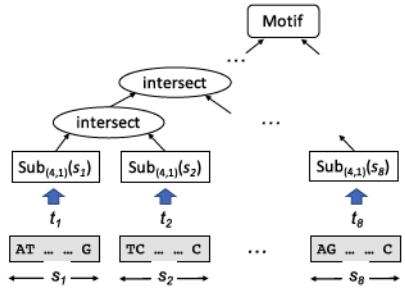


Fig. 2. Recursion tree of applying Algorithm 1 to example shown in Fig. 1.

Motivation. Though previous works claim that parallelization is enabled for the existing sequential EMS solvers by utilizing the above parallel design, the efficient data structures used for sequential EMS cannot be directly used in a parallel setting due to concurrency issues. For example, in [8], a tree based data structure is proposed to store the motif candidates. However, this structure cannot be directly used in the concurrent algorithms. In fact, the parallel versions of the latest EMS solvers [8], [9] are not available for all multi-core machines, due to their extremely high memory consumption.

3 Parallel EMS Algorithms

This section presents two advanced parallel EMS algorithms called recursEMS and parEMS.

3.1 Overview of recursEMS

The recursion-driven parallel framework has been well studied and routinely used for a wide range of applications [12]. Inspired by recursive parallel methods, we propose Algorithm 1 where we find all (l,d)-motifs from input sequences in a recursive manner. If S is empty (Lines 1-3) then Algorithm 1 returns the empty motif set. Otherwise, if S contains just one sequence s (Line 11), then we compute $Sub_{(l,d)}(s)$. For multiple sequences, Algorithm 1 partitions S into two equal-size subsets S_1 and S_2 (Line 5), and then recurs on the two subproblems (Lines 6 and 7). Finally, Algorithm 1 returns the intersection of the results in the two subproblems (in Line 8). Fig. 2 presents the recursion tree of applying Algorithm 1 to the same EMS problem discussed in Fig. 1. Notice that recursEMS utilizes coarse-grain parallelism where each processing unit works on an entire sequence instead of sub-strings of a sequence.

Algorithm 1. recursEMS(S,l,d)

Result: (l,d)-Motifs in the set of strings S
1: **if** $|S| = 0$ **then**
2: return $\emptyset$;
3: **if** $|S| > 1$ **then**
4: Partition the set S evenly into two subsets S_1 and S_2;
5: $C_1 :=$ **fork**(recursEMS(S_1,l,d));
6: $C_2 :=$ **fork**(recursEMS(S_2,l,d));
7: $C := C_1 \cap C_2$;
8: return C;
9: **else**
10: return $Sub_{(l,d)}(s)$, where $s \in S$;
11: // fork a process that calculates $Sub_{(l,d)}(s)$;

Theorem 1 (Soundness). *Algorithm 1 is sound.*

Proof. The soundness of Algorithm 1 means that any string returned by Algorithm 1 is actually an (l,d)-motif of the set S.

We prove that by induction on the levels of the recursion tree, starting from the leaves as the base case. The *base case* of the proof is for the case where there is a single sequence in S. Line 11 correctly computes the set of l-mers of any substring x of length k, where $l-d \leq k \leq l+d$. The soundness of this step is based on the neighborhood generation algorithm of [8], i.e., any string returned in Line 11 appears in s with an edit distance d. Thus, by definition, the strings returned in Line 11 are (l,d)-motifs of the only sequence in S. The *induction hypothesis* states that Algorithm 1 has correctly computed (l,d)-motifs in level j, where $1 \leq j < \log n$, of the recursion tree. In the *inductive step* we prove that (l,d)-motifs are correctly computed in level $j+1$. At level j, we have $n/(2^{j-1})$ sets of l-mers. Then, in level $j+1$, we group the sets each containing at most two sets and take intersection of the sets in every group, creating $n/(2^j)$ sets of l-mers that appear in all sets of level j within a distance d. Therefore, the resulting set in level $\log n$ (i.e., the root) contains all (l,d)-motifs of S. □

Theorem 2 (Completeness). *Algorithm 1 is complete.*

Proof. We show that if there is an (l,d)-motif for the set S, then Algorithm 1 finds it. By contradiction, let x be an (l,d)-motif that Algorithm 1 fails to return. Thus, x must appear in all n sequences in S within an edit distance d. That is, $N_d(x)$ intersects with $(Sub_{(l,d)}(s_1) \cap Sub_{(l,d)}(s_2) \cdots \cap Sub_{(l,d)}(s_n))$. As a result, x should have appeared in each leaf of the recursion tree. This means that x would have climbed up the tree and would have been returned. In other words, if Algorithm 1 doesn't return a string x as an (l,d)-motif, then x is not an (l,d)-motif for S. □

3.2 Complexity Analysis of recursEMS

To examine the benefits brought by recursEMS, we first analyze its worst case asymptotic time complexity and then compare it with the state-of-the-art.

Theorem 3. *The worst case asymptotic time complexity of Algorithm 1 is* $O(mnd^{d+1}l^{d+1}|\Sigma|^d)$.

Proof. At the level of leaves of the recursion tree (Line 11), we need to compute the d-neighbors of each sequence of length m. We use Pal et al.'s approach [8] to compute the d-neighbors. This step of the algorithm has an asymptotic time complexity of $O(md^{d+1}l^d|\Sigma|^d)$ for each leaf. Since we execute all leaves in parallel, the overall time cost of the leaf level remains $O(md^{d+1}l^d|\Sigma|^d)$. Pal et al. [8] state that the cost of intersecting the d-neighborhoods for n sequences is at most $O(mnd^{d+1}l^{d+1}|\Sigma|^d)$. Algorithm 1, however, intersects leaves in a pairwise fashion. Thus, the time complexity of intersecting two leaves is $O(md^{d+1}l^{d+1}|\Sigma|^d)$. Since the height of the recursion tree in Fig. 2 is at most $\log n$, and in each level i we do at most $n/(2^i)$ intersections, we have $O(n)$ intersection operations in the tree. Therefore, the asymptotic time complexity of Algorithm 1 is $O(mnd^{d+1}l^{d+1}|\Sigma|^d)$. □

Note that, Algorithm 1 does not improve the *asymptotic* time complexity of the current EMS solvers [8], but we may still get experimental benefit because *recursEMS* removes the union operations needed for constructing each $Sub_{(l,d)}(s_i)$. In fact, our experiments (Section 5) show that *recursEMS* provides 94% time efficiency improvement compared with the state-of-the-art algorithms, while it underperforms for 220% in terms of space costs. Next section presents another parallel algorithms that is both time and space efficient.

3.3 parEMS: Parallelization With Introducing Concurrent Data Structure

In this section, we propose a novel design called parEMS, as shown in Algorithm 2 and 3. The rationale behind it is to avoid the redundant storage of same motif candidates found by different processing units and implicit calculation of the intersections performed in recursEMS. We achieve this goal by using a shared hash map for n parallel threads, where all threads can store the l-mers that they generate if it is not already generated by other threads.

Algorithm 2 presents the pseudo code running in n threads as the leaves of the recursion tree. However, it does not explicitly perform any set intersections. Instead, we use a shared concurrent hash table that is a key-value storage system (Line 1 in Algorithm 2). The key is the generated l-mer and the value is a bitset of length n. For a given l-mer x, the kth bit in its bitset (where $1 \leq k \leq n$) is set to 1 *iff* thread k has generated x. Similar to recursEMS, each thread j utilizes the algorithm of [8] for the generation of d-neighbors (Line 2-3) of the jth sequence in S. The while loop in Line 4 processes each l-mer as long as there are such strings. Specifically, thread j checks whether the generated l-mer x is already in the concurrent hash table. If that is the case (Lines 9-10), then that means another thread has already generated and stored x. In this case, we just set the jth bit of the bitset associated with x in order to indicate that thread j has also generated x. Otherwise, thread j inserts x and sets the jth bit of the associated bitset. A thread terminates when there are no more l-mers. When all threads have terminated, Algorithm 3 searches through the hash table and returns every l-mer whose associated value (i.e., bitset) is equal to a string of only 1 s. Each bitset indicates that its key (i.e., l-mer) has been generated by all threads. That is, such l-mers are in the intersection of all d-neighborhoods.

Algorithm 2. Thread j in parEMS

Input: $ds[j]$: sequences in S associated with thread j
Result: (l,d)-Motifs of $ds[j]$ generated by thread j.
1: KeyValuePointer *kv*;
2: dNHood := generate-d-neighbors($ds[j]$);
3: *lmer* := next-d-neighbor(dNHood);
4: **while** (*lmer* $\neq$ *NULL*) **do**
5: kv = *con_hash_table*.find (*lmer*);
6: **if** (*kv* = *NULL*) **then**
7: *con_hash_table*.insert (*lmer*,kv $\rightarrow$ Value.set(j));
8: **else**
9: (kv $\rightarrow$ Value).set(j);
10: *lmer* := next-d-neighbor(dNHood);

Algorithm 3. Extract_Motifs

Input: *con_hash_table:* the concurrent hash table used.
Result: (l,d)-Motifs of the set S.
1: keyValuePointer kv;
2: kv := *con_hash_table*.begin();
3: $M := \emptyset$;
4: **while** (*kv* $\neq$ *NULL*) **do**
5: **if** ((*kv* $\rightarrow$ *Value*) = 1^*) **then**
6: // All bits are equal to 1;
7: $M = M \cup$ {kv $\rightarrow$ Key};
8: kv = *con_hash_table*.next();
9: return M;

Theorem 4. *The asymptotic time complexity of parEMS method (Algorithms 2 and 3) is* $O((md^{d+1}l^d|\Sigma|^d) + \alpha|\Sigma|^l)$, *where* α *denotes the load factor of the hash table.*

Proof. Upon the discovery of an l-mer x, thread j stores x in the concurrent hash table and/or sets the jth bit of the bit vector associated with x. The expected time complexity of *insert*, *find* and *set* operations is $O(\alpha)$, where α denotes the load factor of the concurrent hash table. The worst case number of iterations of the for-loop in Line 4 of Algorithm 2 depends on the worst case number of l-mers generated for each sequence, which is $O(|\Sigma|^l)$. Thus, the

TABLE 1
Implementations of RecursEMS and ParEMS

	recursEMS	parEMS
Programming Model	Processes/Tasks	Threads
Shared Memory Access	Serialized	Concurrently
Data Structure used	Tree	Hash Map
Creation Overheads	Heavy	Lightweight

asymptotic cost of interacting with the hash table is $O(\alpha|\Sigma|^l)$. The complexity of generating d-neighbors of a sequence is $O(md^{d+1}l^d|\Sigma|^d)$ (according to [8]). Ideally, if each thread runs on an independent core in a parallel fashion, then the time complexity of generating d-neighbors of all n sequences has an upper bound of $O(md^{d+1}l^d|\Sigma|^d)$. Therefore, the overall asymptotic time complexity of parEMS is $O((md^{d+1}l^d|\Sigma|^d) + \alpha|\Sigma|^l)$. □

4 Implementation

We use C++ to implement the algorithms of Section 3 in a toolset called Scalability-sensitive Parallel EMS (SPEMS) solver. SPEMS provides a uniform interface to solve various EMS instances via recursEMS or parEMS. The major arguments to the interface include the (l, d) instances, the path to targeted biological data, and the performance or memory usage expectation, i.e., best performance or sub-optimal performance but possible to be space efficient. If high space efficiency is desired and the number of sequences is smaller than a predefined threshold, parEMS will be selected. The interface also supports explicitly choosing a parallel scheme. The number of cores used is automatically configured – we choose the smaller number between the number of available cores and the number of input sequences.

For the implementation of recursEMS in SPEMS, we utilize the EMS solver in [8] to generate the motif candidates for a single sequence (line 11 in Algorithm 1). Considering that the EMS solver (i.e., *EMS2* [8]) provides a tree structure, which is time-efficient but requires huge memory usage in storing intermediate motifs candidates, we choose the multiple processes programming model, instead of thread model (like POSIX Threads), to avoid any memory overflow. Specifically, we use the POSIX compliant system call `fork()` to create a new process run in separate memory spaces. Fig. 2 indicates the communication between nodes, i.e., the intersection, only occurs among children and parents, and the intersections in the same level can be run in parallel. However, to avoid memory overflow, we introduce `mmap` to create a new mapping in the virtual address space for performing intersections in shared memory, and then only enable serialized shared memory access among processes. We implement parEMS by introducing the concurrent hash map provided by Intel Threading Building Blocks (TBB), which permits multiple lightweight threads to concurrently access key-vale pairs. The key is the generated motif candidate (i.e., a string), and the value is a bit set. Table 1 summarizes some features of the implementations. The source code is available at: https://github.com/AutoPalSys/SPEMS.

5 Experimental Results

This section presents the experimental evaluation of SPEMS. We first introduce the experimental setup. Then, we compare the execution time and memory consumption of different parallel algorithms. We also vary the (l, d) instances as well as the number of cores to assess their scalabilities.

5.1 Experimental Setup

We compare SPEMS with the prior solvers EMS2 [8] and its advanced version EMS3 [9]. Though these two solvers claim that the corresponding parallel designs are easily implemented, only sequential implementations are available.[1] All settings in these two solvers follow their papers [8], [9]. We perform all experiments on a machine equipped with Intel Xeon Gold 6248R CPUs (total 48 cores) and 384 GB RAM. All programs are written in C++ and compiled by GCC 7 with the "-O3" optimization flag. We collected execution time and resource usage by utilizing Linux system calls. The results reported are the average of three repetitive runs.

TABLE 2
Datasets Overview

Dataset	#Seq	Length	Dataset	#Seq	Length
hm01r	18	2000	yst08r	11	1000
hm02r	9	1000	egr1	3000	<= 469
hm03r	10	1500	elf1	3000	<= 635
hm04r	13	2000	hnf4	3000	<= 297
hm08r	15	500	myc	3000	<= 305
hm20r	35	2000	nfy	3000	<= 819
hm26r	9	1000	sp1	3000	<= 723
mus02r	9	1000	suf	3000	<= 314
mus11r	12	500	yy1	2077	<= 411
yst01r	9	1000	synt	20	600
yst03r	8	500			

Datasets. The benchmarks are mainly collected from two *categories* of real-world DNA datasets, named as *TRANSFAC* and *ChIP-seq*. The first category *TRANSFAC* [13] contains DNA sequences from three species: human (dataset names with prefix *hm*), mouse (dataset names with prefix *mus*), and Saccharomyces cerevisiae (dataset names with prefix *yst*). The second category *ChIP-seq* [14] consists of eight *Homo sapiens* datasets, with being named after the corresponding transcription factor. We also evaluate SPEMS on a synthetic dataset proposed in [8] to further verify the effectiveness of our work. In such a dataset, a specific motif has been planted in each sequence for testing the ($l = 12$, $d = 2$) instance. All detailed information can be found in Table 2. Note that for *ChIP-seq*, we follow the settings in [9], i.e., if the number of sequences in the original source is larger than 3000, only the first 3000 sequences are used for evaluation.

5.2 Performance: Time and Space

Table 3 reports the execution time and the memory consumption of different algorithms. The workload of each core in our machine is determined by the ratio of the number of sequences to the number of cores. For example, in dataset *hm01r*, there are 18 sequences and thus only 18 cores are used for parallel execution. If the number of available cores is less than the number of sequences, all available cores will be used (i.e., 48 cores in our experiments) in several rounds of execution where in each round each core processes one sequence. We skip the scenario where multiple cores process one input sequence and leave this for future work with introducing finer-grained parallelism (in fact, other levels of finer-grained parallelisms can also be introduced, such as ILP or SIMD vector units). The (l, d) instances evaluated are the preliminary challenging cases reported in [9].

Comparison With the State-of-the-Art. The proposed algorithms recursEMS and parEMS outperform EMS2 and EMS3 in terms of time for both real-world datasets, with reaching 10× geometric mean speedup in recursEMS and 2.2× in parEMS. Both parallel algorithms achieve significant improvements on *ChIP-seq*. However, for the majority of datasets, they both consume more memory. In particular, recursEMS significantly underperforms in terms of space costs when running on *TRANSFAC*, with 255% memory overheads. This is due to the fact that all d-neighbors generated by

1. We evaluated these two EMS solvers by directly using their published artifacts, and the parallel implementations crashed when running over the testing datasets on our machine.

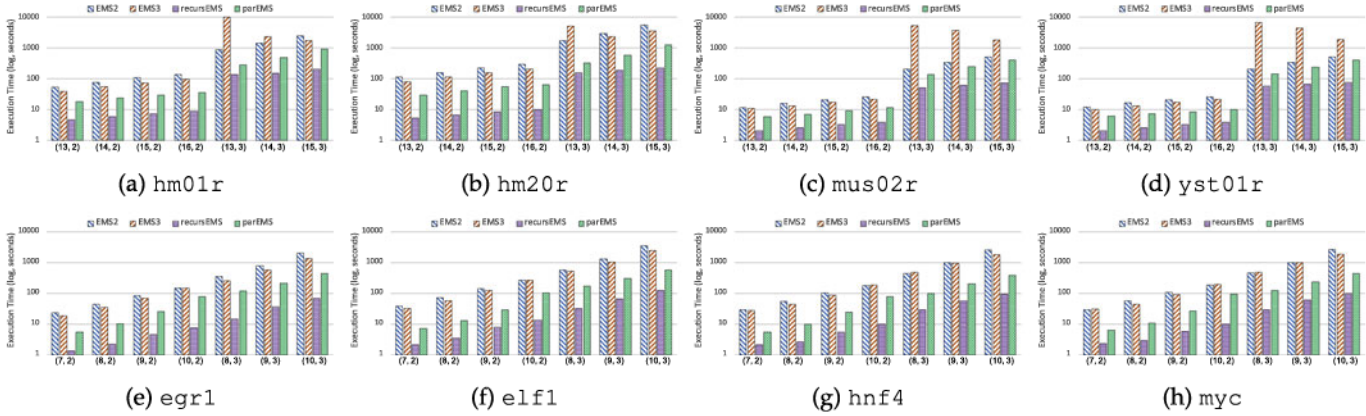
TABLE 3
Performance Comparison: Execution Time and Memory Consumption in Real-World Datasets

Datasets	instance	EMS2	EMS3	recursEMS	parEMS	Datasets	instance	EMS2	EMS3	recursEMS	parEMS
hm01r	(14, 2)	75.97 s 1.17 GB	52.98 s 5.05 GB	**5.81 s** 10.64 GB	25.45 s 4.13 GB	yst03r	(14, 2)	6.13 s 0.32 GB	4.89 s 1.4 GB	**1.17 s** 1.32 GB	3.03 s 0.47 GB
hm02r	(15, 2)	20.48 s 0.84 GB	17.25 s 3.61 GB	**3.21 s** 4.31 GB	8.88 s 1.37 GB	yst08r	(14, 2)	21.01 s 0.64 GB	13.15 s 2.78 GB	**2.63 s** 3.81 GB	8.57 s 1.16 GB
hm03r	(15, 2)	38.71 s 1.26 GB	32.29 s 5.45 GB	**5.30 s** 6.35 GB	13.30 s 1.97 GB	egr1	(9, 2)	94.64 s 18.00 MB	69.28 s 71.83 MB	**4.59 s** 438.14 MB	18.47 s 841.72 MB
hm04r	(14, 2)	52.09 s 1.18 GB	36.21 s 5.11 GB	**5.64 s** 8.31 GB	20.57 s 2.94 GB	elf1	(9, 2)	162.96 s 23.52 MB	110.94 s 97.33 MB	**7.68 s** 638.39 MB	31.66 s 834.28 MB
hm08r	(13, 2)	10.92 s 0.24 GB	7.96 s 1.03 GB	**0.97 s** 1.89 GB	4.41 s 0.78 GB	hnf4	(9, 2)	120.18 s 13.79 MB	87.10 s 75.47 MB	**5.55 s** 559.42 MB	21.34 s 830.45 MB
hm20r	(13, 2)	118.83 s 0.76 GB	77.56 s 3.29 GB	**5.16 s** 14.55 GB	29.58 s 4.91 GB	myc	(9, 2)	124.42 s 19.18 MB	93.01 s 75.47 MB	**5.51 s** 544.12 MB	25.61 s 851.87 MB
hm26r	(15, 2)	21.93 s 0.80 GB	16.46 s 3.46 GB	**3.23 s** 4.24 GB	10.03 s 1.39 GB	nfy	(9, 2)	116.76 s 22.60 MB	87.77 s 98.05 MB	**7.17 s** 490.23 MB	33.48 s 890.75 MB
mus02r	(15, 2)	22.71 s 0.84 GB	16.84 s 3.61 GB	**3.50 s** 4.29 GB	8.77 s 1.18 GB	sp1	(9, 2)	123.08 s 21.00 MB	91.05 s 82.1 MB	**5.94 s** 538.84 MB	20.91 s 890.27 MB
mus11r	(13, 2)	8.23 s 0.23 GB	6.13 s 1.01 GB	**0.95 s** 1.45 GB	3.83 s 0.61 GB	srf	(9, 2)	92.62 s 16.72 MB	66.44 s 65.3 MB	**4.47 s** 453.97 MB	16.83 s 870.68 MB
yst01r	(15, 2)	21.61 s 0.88 GB	17.23 s 3.79 GB	**3.20 s** 4.27 GB	8.37 s 1.16 GB	yy1	(9, 2)	112.77 s 26.51 MB	84.03 s 140.84 MB	**6.47 s** 452.27 GB	30.04 s 650 MB

different processes are stored in memory, where there may be a lot of redundant l-mers. Moreover, the inputs and outputs of the intersection operations in recursEMS should also be stored. Though parEMS presents efficient memory usage when running on the first category, it raises the space cost for *ChIP-seq*, up to 5919%. The memory cost explosion comes from the bitsets stored in the concurrent hash map. For example, in dataset *egr1*, every motif candidate stored in the hash map needs to match to a 3000-bits value because we have three thousand threads. When running different algorithms on the synthetic dataset, we observe that recursEMS and parEMS bring 10.6× and 2.4× speedup, respectively (compared with EMS2). The results are similar to the ones on dataset *TRANSFAC*. All algorithms generate the same motif outputs. In general, EMS2 and EMS3 have significantly lower space costs than the proposed methods. This is to some extent predictable because recursEMS and parEMS are parallel methods with multiple threads or processes, whereas EMS2 and EMS3 are single-thread/process sequential programs, which naturally consume less memory. But note that, the multi-threads/processes programs are still possible to reach similar memory cost by further investing fine-grained management in recursEMS and parEMS.

recursEMS Versus parEMS. Comparing the two proposed parallel methods recursEMS and parEMS, we observe that, on average recursEMS performs 5× better in terms of time costs for the datasets in Category 1, where the length of sequences are large but the number of sequences is smaller than the number of cores. By contrast, when it comes to space costs, parEMS performs 51% better on average. The space cost improvement is due to the use of a concurrent hash table by parEMS, where no redundant l-mers are stored and the extra space needed for the bitset values are limited. As for Category 2, recursEMS outperforms parEMS in terms of both time and space cost (4.1× and 1.6×, respectively). In principle, parEMS has a better asymptotic time complexity compared with recursEMS, as presented in Section 3. However, the overhead from a large number of *find*, *insert* and *set* operations on the concurrent hash map causes the poor timing performance of parEMS. The huge memory usage of parEMS in Category 2, as mentioned above, comes from the extra space used in bitset values. Nonetheless, when considering both time and space costs in a dataset with limited number of sequences, parEMS may provide a more efficient solution (based on our experiments).

Summary. recursEMS has a higher speedup (with respect to the best of EMS2 and EMS3) across the board. The speedups of both recursEMS and parEMS are better for shorter sequences. Moreover, both proposed methods outperform EMS2 and EMS3 in terms of execution time. Overall, recursEMS is the best method in terms of timing.



(a) hm01r (b) hm20r (c) mus02r (d) yst01r

(e) egr1 (f) elf1 (g) hnf4 (h) myc

Fig. 3. Scalability Analysis 1 in Execution Time: Execution Time of Applying Different Algorithms on Eight Representative Datasets under different (l, d) instances.

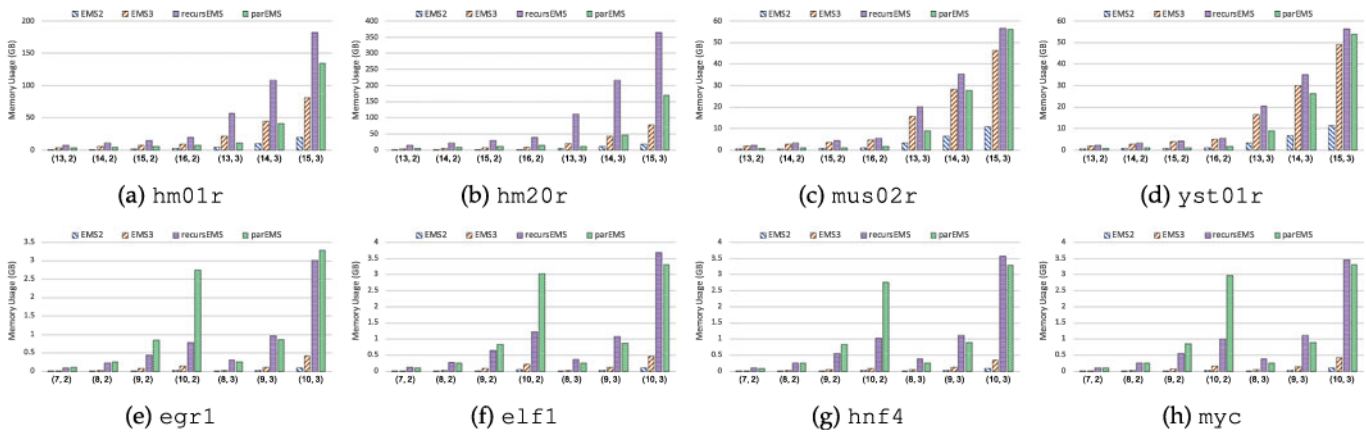


Fig. 4. Scalability Analysis 1 in Memory Usage: Memory Usage of Applying Different Algorithms on Eight Representative Datasets under different (l, d) instances.

5.3 Scalability

This section examines the scalability of the proposed algorithms with respect to the hardness of the EMS instance and the number of utilized cores.

Varying (l, d) Instances. Figs. 3 and 4 show the execution time and memory usage of applying different algorithms to eight representative datasets (we select 4 datasets from each category) under various (l, d) instances. Similar to prior work [8], we only consider the performance over the challenging instances, where $l \in [13, 16]$ and $d \in [2, 3]$.

Overall, recursEMS and parEMS scale well in terms of execution time as the (l, d) instances become harder. For recursEMS, when running on the instances shown in Table 3, the speedups over the sequential EMS2 range from 6.75× to 23.01×, and the speedups over EMS3 range from 5.21× to 16.87×. Fig. 3 illustrates that as the instances become more challenging, the speedup of recursEMS keeps nearly the same when the number of cores used remains constant. For example, in dataset *hm01r*, all speedups are almost around 11.9× in the tested (l, d) instances. The results demonstrate that as the problem size becomes larger, recursEMS can maintain the time efficiency. A similar situation can be found for parEMS to a lesser extent.

Varying the Number of Cores. Fig. 5 illustrates how parEMS and recursEMS scale when we increase the number of cores for a fixed-size instance (i.e., $l = 9$, $d = 2$). We observe that as the number of cores goes beyond 40 the time efficiency of parEMS flattens, whereas recursEMS preserves its efficiency with almost the same rate. Based on our experience, achieving better speedups depends on how efficiently d-neighbors can be generated, which heavily relies on having cost-effective concurrent data structures for storing them.

Summary. For a fixed d, growing l would increase the execution time but not significantly. For a fixed l, increasing d results in significant growth in execution time, especially for long sequences. This is because the complexity of d-neighbor generation is exponential in terms of d (where d is both the base and the exponent of some terms). For a fixed d (respectively, l), increasing l (respectively, d) adds to the space costs significantly because the number of l-mers is increased significantly. As for scalability with respect to the number of processors, the space costs of recursEMS are higher than parEMS in most cases, for both long and short sequences. In the context of datasets with smaller sequences, recursEMS scales much better when increasing the number of cores. parEMS strikes a balance between time and space costs. Thus, if there are memory limitations, then parEMS is a better choice than recursEMS.

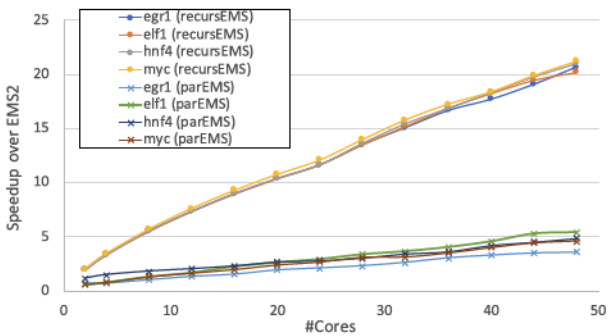


Fig. 5. Scalability Analysis 2: Speedup over EMS2 when applying SPEMS on 4 *ChIP-seq* datasets ($l = 9$ and $d = 2$).

6 Related Work

There exists a rich body of work in finding motifs in biological sequences. Among them, the Planted Motif Search (PMS) and Edited-distance-based Motif Search (EMS) are two critical research directions. Both problems have been identified as NP-Hard [15], and existing solvers mainly fall into three categories: exact, approximate and randomized methods. Exact methods include Davila et al.'s work [16] where they propose multiple PMS algorithms for generating the common neighbors for every pair of l-mers from different strings. Nicolae et al. [3] further improve the performance by using the pruning conditions to efficiently generate neighborhoods for multiple l-mers. Yu et al. [17] design PairMotif to effectively restrict the motif search space. Soundarajan et al. [18] develop a couple of hash-based heuristics to further reduce both the searching space and unnecessary computations used in the traditional tree-based branch and bound mechanisms. On the other hand, instead of collecting all desired motifs, approximate algorithms are developed for fast searching. These algorithms [19], [20] apply heuristic search techniques to avoid some redundant computations while ensuring the search accuracy reaching the required level. In recent years, more flexible algorithms [9], [21], [22] are proposed to allow customized design which consider the balance between output accuracy and computational performance. Randomized algorithms [23] initially form a random subset of the input strings as a small subproblem, and solve the random subproblem using an existing deterministic algorithm. Then, they use the solution of the random subproblem to find the solution of the main problem. Thus, the final results heavily depend on the random subproblem.

In addition to developing efficient motif search algorithms, researchers have also proposed many methods and tools for introducing parallel computing. Perera et al. [24] implement the enhanced brute force algorithm using the POSIX thread library to accelerate motif finding in DNA sequences on multicore CPUs. Abbas et al. also work on the multicore platform, but focused on another exact motif search algorithm, called cVoting [25]. Kongmunvattana [26] studies parameter spaces which include the length of motifs, the number of input sequences, the allowable Hamming distance, and the number of processor cores, to find an optimal point for load balancing between parallel processes when

they are collaboratively and concurrently searching for motifs. Roy et al. [27] solve motif search problems by using streaming execution over a large set of non-deterministic finite automata (NFA) and thus took advantage of an emerging parallel architecture. Kazemian et al. [28] target the decomposition granularity of motif finding algorithm and parallelize a PMS algorithm (qPMS9) on multi-core systems with dynamic scheduling of threads and efficient parallelization of loops using OpenMP library. Salomon et al. [29] introduced a dynamic load balancing method into heterogeneous computation models in which CPU and GPU collaborate to exploit the maximum efficiency on the system.

In contrast with randomized and approximate methods, our approach is an exact method for the EMS problem (instead of PMS). To the best of our knowledge, the state-of-the-art EMS methods include EMS2 [8] and EMS3 [9], and the proposed SPEMS approach significantly outperforms these method on many datasets in the challenging benchmarks *ChIP-seq* and *TRANSFAC*.

7 Conclusion

We proposed two parallel algorithms for Edit distance-based Motif Search, namely recursEMS and parEMS, and integrated them into a scalability-sensitive parallel EMS solver SPEMS. The recursEMS algorithm significantly improves time efficiency of the state-of-the-art when evaluated with respect to two real-world datasets (*TRANSFAC* and *ChIP-seq*) and a synthetic dataset. However, there is room for improving its space efficiency. parEMS remedies this problem by proposing a novel method that avoids storing redundant strings while utilizing a concurrent hash table.

In future work, we will investigate a more efficient concurrent data structure for storing and retrieving strings. For instance, it is desirable to have a concurrent data structure that can work faster but less memory intensive for motifs with wildcards. The other direction that should be complementary with EMS parallelization is to explore vectorization, which in fact introduces finer-grained parallelization.

Acknowledgments

We thank all anonymous reviewers for their constructive comments. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 2105006. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was performed when the first author was an assistant professor at Michigan Technological University.

References

[1] K. Raza, "Application of data mining in bioinformatics," 2012, *arXiv:1205.1125*.
[2] K. Benkrid, P. Velentzas, and S. Kasap, "A high performance reconfigurable core for motif searching using profile HMM," in *Proc. NASA/ESA Conf. Adaptive Hardware Syst.*, 2008, pp. 285–292.
[3] M. Nicolae and S. Rajasekaran, "Efficient sequential and parallel algorithms for planted motif search," *BMC Bioinf.*, vol. 15, no. 1, pp. 1–10, 2014.
[4] W. N. Grundy, T. L. Bailey, and C. P. Elkan, "Parameme: A parallel implementation and a web interface for a DNA and protein motif discovery tool," *Bioinformatics*, vol. 12, no. 4, pp. 303–310, 1996.
[5] T. L. Bailey and C. Elkan, "Unsupervised learning of multiple motifs in biopolymers using expectation maximization," *Mach. Learn.*, vol. 21, no. 1, pp. 51–80, 1995.
[6] J. Qin, S. Pinkenburg, and W. Rosenstiel, "Parallel motif search using parseq," in *Proc. Parallel Distrib. Comput. Netw.*, 2005, pp. 601–607.
[7] M. Ferretti and M. Musci, "Geometrical motifs search in proteins: A parallel approach," *Parallel Comput.*, vol. 42, pp. 60–74, 2015.
[8] S. Pal, P. Xiao, and S. Rajasekaran, "Efficient sequential and parallel algorithms for finding edit distance based motifs," *BMC Genomic.*, vol. 17, no. 4, pp. 315–326, 2016.
[9] P. Xiao, X. Cai, and S. Rajasekaran, "EMS3: An improved algorithm for finding edit-distance based motifs," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 18, no. 1, pp. 27–37, Jan./Feb. 2021.
[10] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 5, pp. 522–532, May 1998.
[11] M. Li, B. Ma, and L. Wang, "On the closest string and substring problems," *J. ACM*, vol. 49, no. 2, pp. 157–171, 2002.
[12] J. Misra, "Powerlist: A structure for parallel recursion," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1737–1767, 1994.
[13] M. Tompa et al., "Assessing computational tools for the discovery of transcription factor binding sites," *Nature Biotechnol.*, vol. 23, no. 1, pp. 137–144, 2005.
[14] P. Kheradpour and M. Kellis, "Systematic discovery and characterization of regulatory motifs in encode TF binding experiments," *Nucleic Acids Res.*, vol. 42, no. 5, pp. 2976–2987, 2014.
[15] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang, "Distinguishing string selection problems," *Inf. Comput.*, vol. 185, no. 1, pp. 41–55, 2003.
[16] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l, d) motif search," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 4, no. 4, pp. 544–552, Oct.–Dec. 2007.
[17] Q. Yu, H. Huo, Y. Zhang, and H. Guo, "Pairmotif: A new pattern-driven algorithm for planted (l, d) DNA motif search," *PLoS One*, vol. 7, no. 10, 2012, Art. no. e48442.
[18] S. Soundarajan, M. Salomon, and J. H. Park, "Efficient branch and bound motif finding with maximum accuracy based on hashing," in *Proc. IEEE 9th Annu. Comput. Commun. Workshop Conf.*, 2019, pp. 0866–0872.
[19] C.-W. Huang, W.-S. Lee, and S.-Y. Hsieh, "An improved heuristic algorithm for finding motif signals in dna sequences," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 8, no. 4, pp. 959–975, Jul./Aug. 2010.
[20] C. Sun, H. Huo, Q. Yu, H. Guo, and Z. Sun, "An affinity propagation-based DNA motif discovery algorithm," *BioMed Res. Int.*, vol. 2015, 2015, Art. no. 853461.
[21] H. Al-Shaikhli and E. de Doncker, "qsmf: An approximate algorithm for quotum planted motif search on chip-seq data," in *Proc. IEEE Int. Conf. Electro Inf. Technol.*, 2019, pp. 434–440.
[22] F. B. Ashraf and M. S. R. Shafi, "Mfea: An evolutionary approach for motif finding in dna sequences," *Informat. Med. Unlocked*, vol. 21, 2020, Art. no. 100466.
[23] P. Xiao, S. Pal, and S. Rajasekaran, "qPMS10: A randomized algorithm for efficiently solving quorum planted motif search problem," in *Proc. IEEE Int. Conf. Bioinf. Biomed.*, 2016, pp. 670–675.
[24] P. Perera and R. Ragel, "Accelerating motif finding in dna sequences with multicore cpus," in *Proc. IEEE 8th Int. Conf. Ind. Inf. Syst.*, 2013, pp. 242–247.
[25] Y. Xu, J. Yang, Y. Zhao, and Y. Shang, "An improved voting algorithm for planted (l, d) motif search," *Inf. Sci.*, vol. 237, pp. 305–312, 2013.
[26] A. Kongmunvattana, "Load balancing for parallel motif discoveries," *Int. J. Comput. Appl.*, vol. 124, no. 13, pp. 29–34, 2015.
[27] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 13, no. 1, pp. 99–111, Jan./Feb. 2015.
[28] F. S. Kazemian, M. Fazlali, A. Katanforoush, and M. Rezvani, "Parallel implementation of quorum planted (l, d) motif search on multi-core/many-core platforms," *Microprocessors Microsystems*, vol. 46, pp. 255–263, 2016.
[29] M. Salomon, S. Soundarajan, and J. H. Park, "PDMF: Parallel dictionary motif finder on multicore and GPU," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun.; IEEE 17th Int. Conf. Smart City; IEEE 5th Int. Conf. Data Sci. Syst.*, 2019, pp. 114–122.

Junqiao Qiu received the bachelor's degree from Sun Yat-sen University, in 2015, and the PhD degree from the computer science and engineering department, University of California, Riverside, in 2020. He is currently an assistant professor of computer science with the City University of Hong Kong. Before joining City University of Hong Kong, he was an assistant professor of computer science with Michigan Technological University. His research interests include parallel computing and high-performance data analytics.

Ali Ebnenasir received the bachelor's and master's degrees from the University of Isfahan and Iran University of Science and Technology, in 1994 and 1998, respectively, and the PhD degree from Computer Science and Engineering Department, Michigan State University (MSU). He is currently an associate professor of computer science with Michigan Technological University and a senior member of the ACM. After finishing his postdoctoral fellowship with MSU, in 2006, he joined the Department of Computer Science, Michigan Tech. His research interests include software dependability, formal methods and parallel and distributed computing.