Parallel Motif Extraction from Very Long Sequences

Majed Sahli
King Abdullah University of Science & Technology
Thuwal, Saudi Arabia
majed.sahli@kaust.edu.sa

Essam Mansour
Qatar Computing Research Institute (QCRI)
Doha, Qatar
emansour@qf.org.qa

Panos Kalnis
King Abdullah University of Science & Technology
Thuwal, Saudi Arabia
panos.kalnis@kaust.edu.sa

ABSTRACT

Motifs are frequent patterns used to identify biological functionality in genomic sequences, periodicity in time series, or user trends in web logs. In contrast to a lot of existing work that focuses on collections of many short sequences, modern applications require mining of motifs in one very long sequence (i.e., in the order of several gigabytes). For this case, there exist statistical approaches that are fast but inaccurate; or combinatorial methods that are sound and complete. Unfortunately, existing combinatorial methods are serial and very slow. Consequently, they are limited to very short sequences (i.e., a few megabytes), small alphabets (typically 4 symbols for DNA sequences), and restricted types of motifs.

This paper presents ACME, a combinatorial method for extracting motifs from a single very long sequence. ACME arranges the search space in contiguous blocks that take advantage of the cache hierarchy in modern architectures, and achieves almost an order of magnitude performance gain in serial execution. It also decomposes the search space in a smart way that allows scalability to thousands of processors with more than 90% speedup. ACME is the only method that: (*i*) scales to gigabyte-long sequences; (*ii*) handles large alphabets; (*iii*) supports interesting types of motifs with minimal additional cost; and (*iv*) is optimized for a variety of architectures such as multi-core systems, clusters in the cloud, and supercomputers. ACME reduces the extraction time for an exact-length query from 4 hours to 7 minutes on a typical workstation; handles 3 orders of magnitude longer sequences; and scales up to 16,384 cores on a supercomputer.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*

Keywords

Motif, Suffix Tree, Parallel, Cache Efficiency, in-memory

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505575.

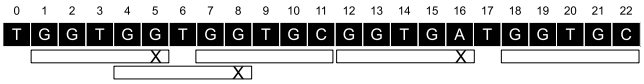


Figure 1: Example sequence S over DNA alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. Occurrences of motif candidate $m = \texttt{GGTGC}$ are indicated by boxes, assuming distance threshold $d = 1$. X refers to a mismatch between m and the occurrence. Occurrences may overlap.

1. INTRODUCTION

Motifs are patterns that appear frequently in sequences. Although there exist numerous methods [17] to extract motifs from a dataset of many short sequences, this paper deals with the more computationally demanding case of a single very long sequence. Many modern applications require motif extraction from one long sequence. Examples include human genome analysis in bioinformatics [21]; stock market prediction in time series [16]; and web log analytics [18]. Such data contain errors, noise, and non-linear mappings [22]. Hence, it is necessary to support approximate matching in motif extraction, meaning that occurrences of a motif may differ slightly from the motif according to a distance function.

Motif extraction approaches are classified into two categories: statistical and combinatorial [5]. Statistical approaches rely on sampling or calculating the probability of motif existence. Such approaches trade accuracy for speed [13]; they may miss some motifs (i.e., false negatives), or return motifs that do not exist (i.e., false positives). Combinatorial approaches [1, 9, 10] verify all combinations of symbols and return all motifs that satisfy the required distance threshold (i.e., no false positives or negatives). This paper focuses on combinatorial motif extraction approaches.

Example. Query q looks for motifs that occur at least $\sigma = 5$ times with a distance of at most $d = 1$ between a motif and an occurrence. Let $m = \texttt{GGTGC}$ be a candidate motif. Figure 1 shows sub-sequences of S that match m. The distance of each occurrence (e.g., GGTGG) from m is at most 1 (i.e., G instead of C at positions 5 and 8). An occurrence is denoted as a pair of start and end positions in S. The set of occurrences for m is $\mathcal{L}(m) = \{(1, 5), (4, 8), (7, 11), (12, 16), (18, 22)\}$ and the frequency of m is $|\mathcal{L}(m)| = 5$.

Compared to the well-studied frequent itemset mining problem in transactional data, motif mining in sequences has three differences: (*i*) Order is important. For example, AG may be frequent even if GA is infrequent. (*ii*) Motif occurrences may overlap. For example, in sequence AAA, the oc-

currences set of motif AA is $\mathcal{L}(\texttt{AA}) = \{(0,1),(1,2)\}$. (*iii*) Because of the distance threshold, a valid motif may not appear as a subsequence within the input sequence. For example, in sequence AGAG, with frequency and distance thresholds $\sigma = 2$ and $d = 1$; TG is a motif. Because of these differences, solutions for frequent itemset mining, such as the FP-tree [12], cannot be utilized. Instead, all combinations of symbols from the alphabet Σ must be checked. Assuming that the length of the longest valid motif is l_{max}, the search space size is $\mathcal{O}(|\Sigma|^{l_{max}})$.

Because of the exponential increase in runtime, existing methods attempt to limit the search space by restricting the motif types that can be identified [13, 15]. FLAME [9], for instance, searches for motifs of a specific length only. Despite this restriction, the largest reported input sequence was only 1.3MB. Another way to limit the search space is by limiting the distance threshold. For example, MADMX [10] introduced a so called density measure and VARUN [1] utilized saturation constraints. Both are based on the idea of decreasing the distance threshold for shorter motifs in order to increase the probability of early pruning. Nevertheless, the largest reported input did not exceed 3.1MB. It must be noted that MADMX and VARUN support only 4-symbol DNA sequences. With larger alphabets (e.g., English alphabet), they would handle smaller sequences in practice, due to the expanded search space.

All aforementioned methods are serial. Supporting larger inputs demands parallel processing. Unfortunately, parallelization of the motif extraction process is not easy. There are two options: (*i*) Partition the input sequence, which requires an expensive merging step since motif candidates must be validated against the entire input sequence. The required communication grows quadratically with the number of processors and limits scalability. (*ii*) Partition the search space and replicate the entire input sequence. This minimizes communication but affects load balance. Because pruning can happen at different depths of the search space that cannot be predicted beforehand. It may happen that only few processors do the majority of the work, while other processors stay idle. To the best of our knowledge[1], there is only one parallel approach, called PSmile [3], that employs heuristic partitioning but scales to only 4 processors. The largest reported input using PSmile is less than 0.25MB.

In this paper, we present ACME, a parallel combinatorial method for extracting motifs from a single very long sequence. ACME handles gigabyte-long sequences, such as the entire human genome (i.e., 2.6GB). Similar to some existing methods, ACME uses a suffix tree [11] to keep occurrence counts for all suffixes in the input sequence S. The novelty of ACME lies in: (*i*) the traversal order of the search space, and (*ii*) the order of accessing information in the suffix tree. These are arranged in a way that they exhibit spatial and temporal locality. This allows us to store the required information in contiguous memory blocks that are kept in the CPU caches, and minimize cache misses in modern processor architectures. In addition, the cached information facilitates fast backtracking, which in turn allows the identification of right-supermaximal motifs (see Section 2.1 for definition) with minimal overhead. By being cache-efficient, ACME achieves almost an order of magnitude performance improvement for *serial* execution.

[1]There exist several parallel approaches [4, 6, 7, 14] for the much simpler case of a collection of many short sequences.

ACME also supports large scale parallelism. It partitions the search space to a large number (i.e., tens to hundreds of thousands) of independent tasks. A master-worker approach is employed to keep all processors busy, achieving good load balance. However, fine-grained partitioning may miss opportunities for early pruning, resulting in more work for many processors. The novelty of ACME lies in the development of a set of heuristics that achieve good tradeoff between load balancing and early pruning. We tested ACME in a variety of architectures, including multi-core shared memory workstations, shared-nothing Linux clusters, and a large supercomputer with 16,384 processors. ACME achieves almost perfect speedup (more than 90%) in most cases.

In summary, our contributions are:

- We develop a cache-efficient search space traversal technique for motif extraction that improves the serial execution time by almost an order of magnitude.
- We propose heuristics to decompose the motif extraction process to fine-grained tasks, allowing for the efficient utilization of thousands of processors. We scale to 16,384 processors on an IBM BlueGene/P supercomputer and solve in 18 minutes a query that needs more than 10 days on a high-end multicore machine.
- Our method scales to large alphabets (e.g., English alphabet for the Wikipedia dataset) and supports interesting motif types (e.g., right-supermaximal motifs) with minimal overhead.
- We conduct a comprehensive evaluation with large real datasets. ACME handles 3 orders of magnitude longer sequences than our competitors on the same machine.

The rest of this paper is organized as follows. Section 2 presents the required background. Related work is discussed in Section 3. We introduce our cache-efficient method and parallel approach in Sections 4 and 5. Section 6 presents the experimental analysis and Section 7 concludes the paper.

2. BACKGROUND

This section introduces necessary definitions and defines the problem. The problem search space and the index used in the solution are then discussed.

2.1 Motifs

A sequence S over an alphabet Σ is an ordered and finite list of symbols from Σ. $S[i]$ is the ith symbol in S, where $0 \leq i < |S|$. A subsequence of S that starts at position i and ends at position j is denoted by $S[i,j]$ or simply by its position pair (i,j). For example, (7,11) represents GGTGC in Figure 1. Let $\mathcal{D}$ be a function that measures similarity between two sequences. Following the previous work [8, 9], in this paper we assume $\mathcal{D}$ is the Hamming distance (i.e., number of mismatches). A *motif candidate* m is a combination of symbols from Σ. A subsequence $S[i,j]$ is an *occurrence* of m in S, if the distance between $S[i,j]$ and m is at most d, where d is a user-defined distance threshold. The set of all occurrences of m in S is denoted by $\mathcal{L}(m)$. Formally: $\mathcal{L}(m) = \{(i,j) | \mathcal{D}(S[i,j], m) \leq d\}$.

DEFINITION 1 (MOTIF). *Let S be a sequence, $\sigma \geq 2$ be a frequency threshold, and $d \geq 0$ be a distance threshold. A candidate m is a* motif *if and only if there are at least σ occurrences of m in S. Formally: $|\mathcal{L}(m)| \geq \sigma$.*

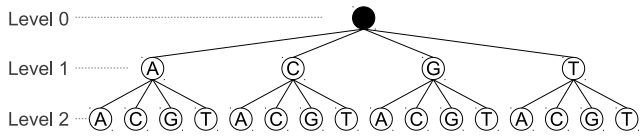


Figure 2: Two levels of the combinatorial search space trie for DNA motifs, alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Definition 2 (Maximal motif). *A motif m is maximal if and only if it cannot be extended to the right nor to the left without changing its occurrences set. Formally, $\mathcal{L}(m) \neq \mathcal{L}(\alpha m) \neq \mathcal{L}(m\beta)$, where α and $\beta \in \Sigma$.*

A maximal motif must be *right* and *left maximal* [8]. A motif m is right maximal if $\mathcal{L}(m\beta)$ has less occurrences or more mismatches than $\mathcal{L}(m)$. Similarly, a motif m is left maximal if extending m to the left causes a loss in occurrences or introduces new mismatches. There is excessive overlapping among maximal motifs. Users are typically interested in longer motifs [8], such as the right-supermaximal ones, denoted by *rs-motifs* in the rest of this paper.

Definition 3 (Right-Supermaximal Motif). *Let M be the set of maximal motifs from Definition 2 and let $\widehat{m} \in M$. $\widehat{m}$ is a* rs*-motif, if $\widehat{m}$ is not a prefix of any other motif in M. We call M_{rs} the set of all rs-motifs. Formally: $M_{rs} = \{\widehat{m} | \widehat{m} \in M, \widehat{m}\alpha \notin M, \alpha \in \Sigma\}$.*

The number of possible motif candidates for a certain σ value is $\sum_{i=1}^{|S|-\sigma+1} |\Sigma|^i$, where $|\Sigma|$ is the alphabet size. To restrict the number of motif candidates, previous works have imposed minimum (l_{min}) and maximum (l_{max}) length constraints.

Problem 1. *Given sequence S, frequency threshold $\sigma \geq 2$, distance threshold $d \geq 0$, minimum length $l_{min} \geq 2$, and maximum length $l_{max} \geq l_{min}$; find all rs-motifs.*

The most interesting case is when $l_{max} = \infty$. Obviously, this is also the most computationally expensive case since the length cannot be used as an upper bound.

2.2 Trie-based Search Space and Suffix Trees

The search space of a motif extraction query is the set of motif candidates for that query. The size of such search space is astronomical even for a short input sequence and a small alphabet. A combinatorial trie (see Figure 2) is used as a compact representation of the search space. Every path label formed by traversing the trie from the root to a node is a motif candidate. Finding the occurrences of each motif candidate and verifying maximality conditions require a large number of expensive searches in the input sequence S. To minimize this cost, a suffix tree index is typically used. A suffix trees is a full-text index where the paths from the root to the leaves correspond to the suffixes of the indexed sequence [11]. It is built in linear time and space as long as the sequence and the tree fit in memory [20].

The properties of the suffix tree facilitate the verification of right and left maximality as discussed by Federico and Pisanti [8]. For the sake of completeness, we highlight the following essential properties. (i) A suffix tree node is *left-diverse* if at least two of its descendant leaves have different left symbols in S. Based on the suffix tree, a motif m is left maximal if one of its occurrences is a left-diverse suffix tree node. (ii) The labels of the children of an internal suffix tree node start with different symbols. Hence, if a motif has an occurrence that consumes the complete label of an internal node, then it is right maximal.

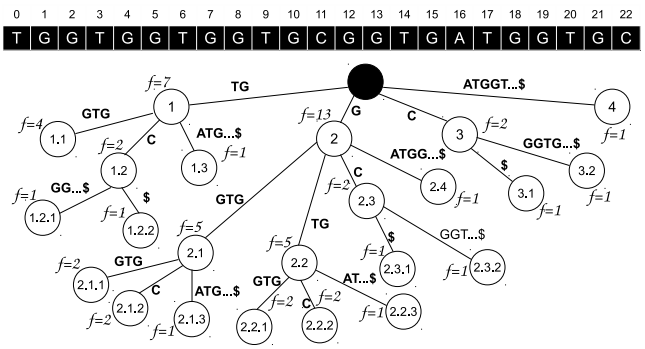


Figure 3: Example sequence over $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and its suffix tree. Suffix tree nodes are annotated with the frequency of their path labels and are numbered for referencing in the paper.

We annotate the suffix tree by traversing it once and storing in every node whether it is left-diverse or not and the number of leaves reachable through it. This number is the frequency of a node's path label. For example, the path label for node `1.2` in Figure 3 is `TGC` and it is not a left-diverse node as `TGC` is always preceded by `G` in S. Node `1.2` is annotated with $f = 2$ because `TGC` appears in S at $\{(9, 11), (20, 22)\}$. For simplicity, Figure 3 does not show the left-diversity annotation. In case of exact motifs, where $d = 0$, the search space is reduced to the suffix tree [2]. For the general case, where $d > 0$, occurrences of a candidate motif are found at different suffix tree nodes. The frequency of the motif is calculated by summing the annotations from all these nodes.

Example. Let us start a depth first traversal of the search space trie in Figure 2 to extract motifs from the example sequence and its suffix tree in Figure 3. Assume $d = 1$ and $\sigma = 10$. The trie traversal starts at node `A` in the first level. By traversing the suffix tree, we find that the first symbol from every branch starting at the root is an occurrence of `A` within distance 1. Therefore, the occurrences set contains the following suffix tree nodes: $\mathcal{L}(\mathtt{A}) = \{1, 2, 3, 4\}$ and a total frequency of $7 + 13 + 2 + 1 = 23$. Search space traversal continues to the first child of `A`, representing the motif candidate `AA`. The occurrences set of `A` is used to create `AA`'s. The label of suffix tree node `1` is `TG`. It already has distance 1 from `A` in the first level. Extending the occurrence by one symbol introduces another mismatch for `AA` so it is discarded. To extend the label for the second occurrence we need to check all branches from suffix tree node `2`. The first three children `2.1` to `2.3` of node `2` are pruned for exceeding the allowed distance. Node `2.4` is added to the occurrences set of `AA` since its path label is `GA`, which has distance 1 from `AA`. The rest of the occurrences of `A` are extended and validated in the same manner. The occurrences set of `AA` is $\mathcal{L}(\mathtt{AA}) = \{2.4, 4\}$ with a total frequency of $1 + 1 = 2$. `AA` is not frequent enough and the search space is pruned by

Table 1: Comparison of combinatorial motif extractors

	Supported motif types					
	Index	Exact-length	Maximal	RS-Motif	Largest reported input	Parallel
FLAME [9]	Suffix Tree	Yes	No	No	1.3 MB	No
VARUN [1]	N/A	No	Yes	No	3.1 MB	No
MADMX [10]	Suffix Tree	No	Yes	No	0.5 MB	No
PSmile [3]	Suffix Tree	Yes	No	No	0.2 MB	Yes
ACME [our method]	Suffix Tree	Yes	Yes	Yes	2.6 GB	Yes

backtracking to node `A` in Figure 2. Then, `AC` is processed in the same way.

3. RELATED WORK

This section presents the most recent combinatorial methods for extracting motifs from a single sequence. Table 1 summarizes these methods. Motif extraction is a highly repetitive process making it directly affected by cache alignment and memory access patterns. For a motif extractor to be scalable, it needs to utilize the memory hierarchy efficiently and run in parallel. Existing methods do not deal with these issues. Therefore, they are limited to sequences in the order of a few megabytes [15].

The complexity of motif extraction grows exponentially with the motif length. Intuitively, extracting maximal motifs and rs-motifs is more complex than extracting exact-length motifs. FLAME [9] supports only exact-length motifs. To explore a sequence, users need to run multiple exact-length queries. VARUN [1] and MADMX [10] support maximal motifs, which cover all motif lengths for the same query thresholds. To limit the search space, VARUN and MADMX define the distance threshold as a ratio with respect to motif candidate length. Both techniques return highly redundant results since they do not support rs-motifs. ACME efficiently extracts exact-length motifs, maximal motifs, and rs-motifs from a long sequence.

Parallelizing motif extraction attracted a lot of research efforts, especially in bioinformatics [3, 4, 6, 7, 14]. None of these approaches extract accurate motifs from a single sequence of gigabyte size. Moreover, most of these approaches are statistical. Challa and Thulasiraman in [4] handled a dataset of 15,000 protein sequences with the longest sequence being 577 symbols only. However, this method did not manage to scale to more than 64 cores. Dasari et al. in [6] extracted common motifs from 20 sequences of a total size of 12KB and scaled to 16 cores. This work was extended in [7] to support GPUs and scaled to 4 GPU devices using the same dataset. Liu, Schmidt, and Maskell in [14] processed a 1MB dataset on 8 GPUs.

The only parallel and combinatorial method for extracting motifs from a single sequence is PSmile [3]. This method attempted to parallelize the motif extraction process by developing a heuristic partitioning approach. The workload of the produced partitions is not equal since they are pruned at different rates. PSmile suffers from highly imbalanced workload and parallel overhead. Moreover, PSmile does not provide any guarantees for the size of the produced partitions [19] and reported scaling to 4 compute nodes only. ACME overcomes this problem by decomposing the search space into fine-grained sub-tries that are dynamically assigned based on their actual workload.

4. CACHE AWARE MOTIF EXTRACTION

ACME decomposes the search space trie into sub-tries of arbitrary sizes. Each sub-trie is maintained and validated independently using a cache-aware mechanism, called CAST[2], presented in this section. Section 5 presents our decomposition and load balancing technique.

4.1 Spatial and Temporal Memory Locality

Recent motif extraction methods realize the trie search space as a set of nodes, where each node has a label of one character and pointers to its parent and children. Additionally, each node contains its occurrences set. These nodes are dynamically allocated and deallocated once they are not needed. The maximum number of trie nodes to be created and then deleted from main memory is $\sum_{i=1}^{l_{max}} |\Sigma|^i$. For example, when l_{max}=15 and $|\Sigma|$=4, the maximum number of nodes is 1,431,655,764. Moreover, These nodes are scattered in main memory and visited back and forth to traverse all motif candidates. Consequently, these methods suffer dramatically from cache misses, plus the overhead of memory allocation and deallocation.

A branch of trie nodes represents a motif candidate (sequence of symbols). These symbols are conceptually adjacent with preserved order; allowing for spatial locality. Moreover, maintaining occurrences sets is a pipelined process, i.e., the occurrences set of `AA` is used to build the occurrences set of `AAA`. That could lead to temporal locality. The existing approaches overlooked these spatial and temporal locality properties in the motif extraction process and the allocation/deallocation overhead.

We propose contiguous data structures to realize the sub-tries and occurrences sets. CAST models the sub-trie search space as a set of variable depth branches represented by their path labels from the root to a leaf. We decouple the sub-trie branches from the occurrences sets to produce smaller entities. Hence, we can benefit from the spatial locality of the branches and the temporal locality of occurrences sets.

For spatial locality, CAST utilizes an array of symbols to recover all branches sharing the same prefix. The size of this array is proportional to the length of the longest motif to be extracted. For instance, a motif of 1K symbols requires roughly a 9KB array. In practice, motifs are shorter. We have experiments with human genome, protein, and English sequences of gigabyte sizes, where the longest motif lengths are 28, 95 and 42 symbols respectively. Moreover, the occurrences set is also realized as an array. With current CPU cache sizes, not only a sub-trie branch will fit in the cache but most probably its occurrences array, too.

For temporal locality, once we maintain the occurrences array $\mathcal{L}(v_i)$ of branch node v_i, we expand each occurrence to create $\mathcal{L}(v_{i+1})$. The upper limit $\mathcal{U}$ of the total frequency of

[2]CAST stands for cache aware search space traversal model

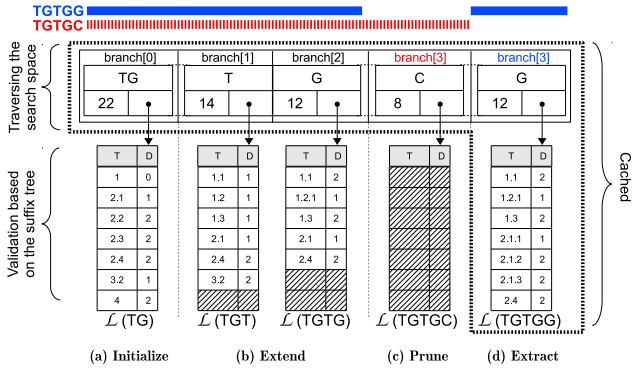


Figure 4: Snapshots of CAST processing for σ=12, l_{min}=l_{max}=5, d=2 over the sequence in Figure 3. Prefix TG is extended one symbol at a time to maintain TGTGC and TGTGG branches. A branch is traversed from ancestor to descendant by moving from left to right. CAST array (*branch*) and the occurrences array of the deepest descendant are easily cached, since both fit into small contiguous memory blocks.

$\mathcal{L}(v_{i+1})$ is the total frequency of $\mathcal{L}(v_i)$. Our method decrements $\mathcal{U}$ by the frequency of discarded elements. Therefore, we can stop as soon as $\mathcal{U} < \sigma$, where σ is the frequency threshold. Hence, CAST achieves high cache efficiency for the traversal and validation processes. Moreover, CAST does not traverse the suffix tree from root, since it keeps a direct pointer to required nodes in occurrences arrays.

4.2 The CAST Algorithm

The CAST algorithm for extracting motifs is illustrated in Algorithm 1. CAST: (*i*) initializes the sub-trie prefix; then (*ii*) extends the prefix as long as it leads to valid motif candidates; otherwise (*iii*) prunes the extension. A query with $\sigma = 12$, $l_{min} = l_{max} = 5$ and $d = 2$ is used to demonstrate Algorithm 1 against sequence S, which is shown in Figure 3.

Algorithm 1 denotes the sub-trie branch array as *branch*. An element *branch*[*i*] contains a symbol c, an integer F, and a pointer, as shown in Figure 4. Each sub-trie has a prefix p that is extended to recover all motif candidates sharing p. *branch*[*i*] represents the motif candidate m_i=$pc_1 \ldots c_i$, where c_i is a symbol from the i^{th} level in the sub-trie (see Figure 2). F_i is the total frequency of m_i and the pointer refers to $\mathcal{L}(m_i)$. Each occurrence in $\mathcal{L}(m_i)$ is a pair $\langle T, D\rangle$, where T is a pointer to a suffix tree node whose path label matches the motif candidate m_i with D mismatches. *branch*[0] represents the fixed-length prefix of the sub-trie. F_0 is a summation of the frequency annotation from each suffix tree node in $\mathcal{L}(p)$.

4.2.1 Prefix Initialization

Algorithm 1 starts by creating the occurrences array of the given fixed-length prefix before recovering motif candidates. CAST commences the occurrences array maintenance for a prefix by fetching all suffix tree nodes at depth one. The maximum size of the occurrences array at this step is $|\Sigma|$. The distance is maintained for the first symbol of the prefix. Then, the nodes whose distances are less than or equal to d are navigated to incrementally maintain the entire prefix.

Input: l_{min}, l_{max}, prefix p
Output: Valid motifs with prefix p

```
1  Let branch be an array of size l_max − |p| + 1
2  branch[0].L ← GETOCCURRENCES(p)
3  branch[0].F ← GETTOTALFREQ(branch[0].L)
4  i ← 1
5  next ← DEPTHFIRSTTRAVERSE(i)
6  While next ≠ END do
7      branch[i].C ← next
8      branch[i].F ← branch[i − 1].F
9      foreach occurrence in branch[i − 1].L do
10         if occurrence is a full suffix tree path label then
               // check child nodes in suffix tree
11             foreach child of occurrence.T do
12                 if first symbol in child label ≠ next then
13                     child.D = occurrence.D + 1
14                     if child.D > d then
15                         Discard(child)
16                         if branch[i].F < σ then
17                             PRUNE(branch[i])
18                 else
19                     add child to branch[i].L
20         else
               // extend within label in suffix tree
21             if next symbol in occurrence.T label ≠ next
               then
22                 increment occurrence.D
23                 if occurrence.D > d then
24                     Discard(occurrence)
25                     if branch[i].F < f then
26                         PRUNE(branch[i])
27             else
28                 add occurrence to branch[i].L
29     if ISVALID(branch[i]) then OUTPUT(branch[i])
30     increment i
31     next ← DEPTHFIRSTTRAVERSE(i)
```

Algorithm 1: CAST MOTIFS EXTRACTION

The number of phases to maintain the occurrences array of prefix p is at most $|p|$.

For example, the sub-trie whose prefix is TG is initialized by CAST in two phases using the suffix tree shown in Figure 3. Figure 4(a) shows the final set of occurrences $\mathcal{L}$(TG) in S. The first element in $\mathcal{L}$(TG) is $\langle 1, 0\rangle$ because the path label of suffix tree node 1 is TG with no mismatches from our prefix. The second element in $\mathcal{L}$(TG) is $\langle 2.1, 1\rangle$ because the first two symbols from the path label of suffix tree node 2.1 are GG with one mismatch from our prefix. The total frequency of TG at *branch*[0] is the frequency annotations from the suffix tree nodes in $\mathcal{L}$(TG), in the same order, 7+5+5+2+1+1+1=22.

4.2.2 Extension, Validation and Pruning

Since TG is frequent enough, it is extended by traversing its search space sub-trie. The depth-first traversal of the sub-trie starts at line 5 in Algorithm 1 to extend *branch*[0]. The extension process considers all symbols of Σ at each level in a depth-first fashion. At level i, DEPTHFIRSTTRAVERSE returns c_i to extend *branch*[$i-1$]. Figure 4(b) demonstrates the extension of *branch*[0] with a T then a G.

The maintenance of occurrences set $\mathcal{L}$ is a pipelined function, where $\mathcal{L}$(*branch*[i+1]) is constructed from $\mathcal{L}$(*branch*[i]).

This process is done in the $foreach$ loop starting at line 9 of Algorithm 1. For example, $\mathcal{L}$(TGT) is created by navigating each element in $\mathcal{L}$(TG). The first element of $\mathcal{L}$(TG) adds suffix tree nodes 1.1, 1.2, and 1.3 to $\mathcal{L}$(TGT) with distance 1 since their labels do not start with T. The second element of $\mathcal{L}$(TG) is added to $\mathcal{L}$(TGT) since its label was not fully consumed. In node 2.2, the next symbol of its label introduces the third mismatch. Thus, the third element of $\mathcal{L}$(TG) is discarded. The rest of $\mathcal{L}$(TG) is processed in the same way. The total frequency at $branch[1]$ drops to 14. Similarly, $\mathcal{L}$(TGTG), $\mathcal{L}$(TGTGC) and $\mathcal{L}$(TGTGG) are created.

A node at $branch[i]$ can be skipped by moving back to its parent at $branch[i-1]$, which is physically adjacent. Therefore, our pruning process has good spatial locality, where backtrack means move to the left. For example in Figure 4(c), the total frequency of TGTGC drops below the frequency threshold σ=10 after discarding node 1.1 of frequency 4 from $\mathcal{L}$(TGTG), i.e. $12-4<\sigma$. Since TGTGC has a frequency less than σ, we do not need to check the rest of the occurrences and the branch is pruned. The if statement at line 16 of Algorithm 1 deals with such cases.

After pruning TGTGC, CAST backtracks to $branch[2]$ which will be extended now using G. All occurrences from $branch[2]$ are also valid for TGTGG at $branch[3]$ with no change in total frequency. The if statement at line 29 of Algorithm 1 returns $true$ since the branch represents a valid motif of length 5 and the function Output is called. The next call to DepthFirstTraverse will find that $i > l_{max}$ so it will decrement i until the level where an extension in the sub-trie is possible or the sub-trie is exhausted.

ACME supports exact-length motifs, maximal motifs, and rs-motifs. Function IsValid in line 29 determines whether a branch represents a valid motif or not as discussed in Section 2. For exact-length motifs, only branches of that length are valid. For maximal motifs IsValid returns $false$ if (i) $branch[i]$ could be extended without changing its occurrences list (i.e., not right maximal) or (ii) none of its occurrences is a left-diverse node (i.e., not left maximal). For rs-motifs, IsValid returns $false$ as long as a maximal motif is frequent and can be extended.

5. PARALLEL MOTIF EXTRACTION

This section presents our efficient parallel tree traversal method (FAST[3]), which partitions horizontally the search space and achieves scalability to thousands of compute nodes. A high degree of concurrency is achieved with minimum communication overhead and a balanced workload across compute nodes.

The key sources of parallel overhead are: (i) contention for the underlying shared resources; (ii) communication overhead; (iii) imbalanced workload leading to lower utilization of available resources; and (iv) redundant and useless work due to lack of appropriate global knowledge. It is challenging to completely avoid these conflicting overhead sources in parallel motif extraction in order to scale to thousands of compute nodes.

5.1 Horizontal Search Space Partitioning

A large trie can be split into numerous sub-tries, where each sub-trie is traversed independently. Parallelizing the trie traversal is easy in this sense. However, the motif extraction search space is pruned at different levels in the trie during the traversal and validation process. Therefore, the workload of each sub-trie is not known in advance. The absence of such knowledge makes load balancing challenging to achieve. Imbalanced workload means a longer makespan affecting the efficiency of parallel systems by underutilizing resources.

[3]FAST stands for fine-grained adaptive sub-tasks

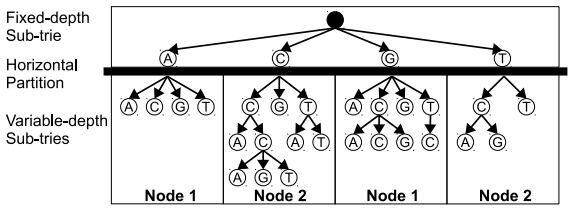


Figure 5: A DNA combinatorial trie partitioned at depth one into a fixed-depth sub-trie leading to four variable-depth sub-tries, which are traversed simultaneously by two compute nodes.

FAST decomposes the search space trie into a large number of independent sub-tries. Our target is to provide enough sub-tries per core to utilize the computing resources with minimal idle time. We partition horizontally the search space trie at a certain depth into a fixed-depth sub-trie and a set of variable-depth sub-tries, as shown in Figure 5. Since the motif search space is a combinatorial trie, there are $|\Sigma|^{l_p}$ sub-tries, where Σ is an alphabet and l_p is a certain depth in the trie (prefix length). The variable-depth sub-tries are of arbitrary size and shape because of pruning motif candidates at different levels. The fixed-depth sub-trie indexes $|\Sigma|^{l_p}$ prefixes. Each prefix is common to a set of motif candidates indexed by a variable-depth sub-trie.

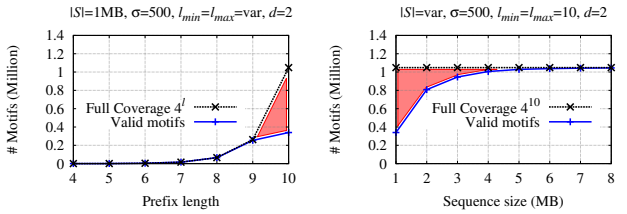
Example. Consider the search space for extracting motifs of length exactly 15 from a DNA sequence ($|\Sigma|=4$). The search space trie consists of 4^{15} (1 Giga) different branches, where each branch is a motif candidate of length 15. If we choose to set our horizontal partition at depth 2, our prefixes will be of length 2 and there are 16 large variable-depth sub-tries. Each sub-trie consists of more than 67 million branches. If the horizontal partition cuts at depth 8 then there are 65,536 independent and small variable-depth sub-tries of 16 thousand branches each.

5.2 The Prefix Length Trade-off

The fixed-depth sub-trie indexes a set of fixed-length prefixes. Each prefix is extended independently to recover a set of motif candidates sharing this prefix. A false positive prefix is a prefix of a set of false positive candidates, which would be pruned if a shorter prefix was used. For instance, if $|\Sigma|=4$ and AA is a prefix that leads to no valid motifs then partitioning the search space using a prefix length of 5 (i.e. horizontally partitioning at depth 5) introduces 64 false positive prefixes that start with AA. The longer the prefix length is, the higher degree of concurrency is achieved. However, enlarging the prefix length increases the probability of having false positive prefixes, which are useless overhead.

Observation 1. *Given distance threshold d, all branches of length d are valid prefixes.*

Any subsequence of length l from the input sequence S will not exceed the distance threshold d for all search space



(a) Variable motif length (b) Variable sequence size

Figure 6: Search space coverage in a DNA sequence. The shaded regions emphasize false positive prefixes, which increase by increasing the prefix length and decrease by increasing the input sequence size.

branches of length l as long as $l \leq d$. For example, if a user allows up to 4 mismatches between a motif candidate and its occurrences then any subsequence of length 4 from the input sequence is a valid occurrence of any prefix of length 4 in the search space. Observation 1 means that no pruning can be done until depth d of the search space, assuming the frequency threshold is met. We say that the search space is *fully covered* at depth d.

OBSERVATION 2. *As the input sequence size increases, the depth of the search space with full coverage increases, too.*

A longer sequence over a certain alphabet Σ means more repetitions from Σ. Therefore, the probability of finding occurrences for motif candidates increases. Our experiments show that even for a relatively small input sequence, the search space can be fully covered to depths beyond the distance threshold. Figure 6(a) shows that prefixes of length less than 10 symbols are fully covered although the sequence size is 1MB. In this experiment, the prefix of length 10 leads to more than 0.5M false positive prefixes, i.e., useless sub-tasks will be processed. Moreover, increasing the size of the input sequence increases the coverage of the search space. Figure 6(b) shows that the number of false positive prefixes generated at l_p=10 in the 1MB sequence decreases by increasing the sequence size.

OBSERVATION 3. *If the search space is horizontally partitioned at depth l_p, where the average number of sub-tries per core leads to high resource utilization, then a longer l_p is not desirable to avoid the overhead of false positives.*

5.3 The FAST Algorithm

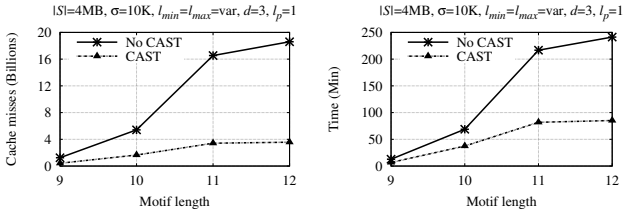
FAST guarantees enough independent sub-tries per core. Each sub-trie is transferred in the compact form of a fixed-length prefix. The generation and distribution cost of a sub-trie is negligible compared to the average computation load per sub-trie. As a preprocessing step, every worker loads the input sequence and constructs its suffix tree in memory to limit overall communication. Dynamic scheduling distributes sub-tries based on their actual workloads. FAST horizontally partitions the search space trie and schedules sub-tasks as shown in Algorithm 2. Fixed-length prefixes are generated serially by the master process. Function GETOPTIMALLENGTH in line 1 of Algorithm 2 calculates the near-optimal prefix length based on Equation 1.

$$l_p = \lceil \log_{|\Sigma|}(\mathcal{K} \times \mathcal{C}) \rceil \quad (1)$$

Input: Alphabet Σ, Number of cores $\mathcal{C}$
Output: Generate and schedule sub-tasks

```
   // Calculate optimal prefix length
1  l_p ← GetOptimalLength( )
2  i ← 0 // An iterator over all prefixes of length l_p
   // Assign sub-tasks
3  While i ≠ prefixes end do
4  |  sub-task ← GetNextPrefix(i)
5  |  WaitForWorkRequest( )
6  |  SendToRequester(sub-task)
7  |  i ← i + 1
   // Signal workers to end
8  While worker exist do
9  |  WaitForWorkRequest( )
10 |  SendToRequester(end)
```

Algorithm 2: PARTITIONINGANDSCHEDULING



(a) L1 cache misses (b) Time performance

Figure 7: Correlation between caches misses and motif extraction time; DNA dataset.

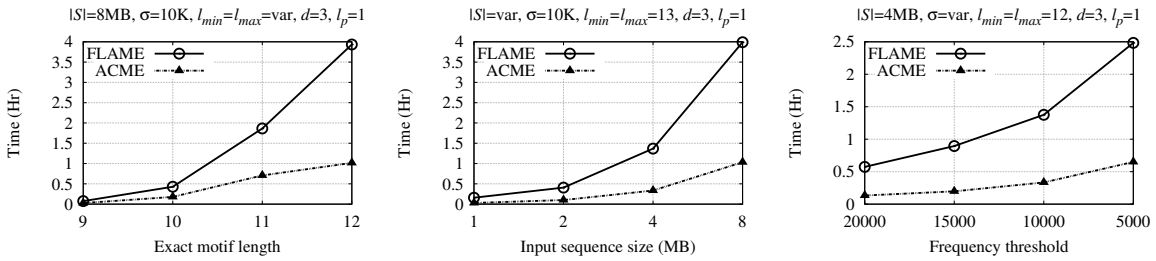
Equation 1 calculates the prefix length (l_p) based on alphabet size $|\Sigma|$ and number of cores $\mathcal{C}$. The near-optimal workload balance is achieved when the average number of sub-tries per core is more than $\mathcal{K}$, i.e., $|\Sigma|^{l_p}/\mathcal{C} > \mathcal{K}$. The exact-length prefixes are generated by a depth-first traversal of the fixed-depth sub-trie. An iterator is used to recover these prefixes by a loop that goes over all combinations of length l_p from Σ. The master process is idle as long as all workers are busy. Algorithm 2 is lightweight compared to the extraction process carried out by workers. Hence, parallelizing the prefix generation does not lead to any speedup in the overall process.

6. EVALUATION

ACME [4] is implemented in C++ with two versions: (*i*) ACME-MPI uses MPI to run on shared-nothing systems, such as clusters and supercomputers. (*ii*) ACME-THR utilizes shared-memory threads on multi-core systems, where the sequence and its suffix tree are shared among all threads; therefore, it can process longer sequences.

We used real datasets of different alphabets: (*i*) DNA[5] from the entire human genome (2.6GB, 4 symbols alphabet); (*ii*) Protein[6] sequence (6GB, 20 symbols); and (*iii*) English[7] text from a Wikipedia archive (1GB, 26 symbols). In some

[4]ACME code and the used datasets are available online at: http://cloud.kaust.edu.sa/Pages/acme_software.aspx
[5]http://webhome.cs.uvic.ca/ thomo/HG18.fasta.tar.gz
[6]http://www.uniprot.org/uniprot/?query=&format=*
[7]http://en.wikipedia.org/wiki/Wikipedia:Database_download



(a) Variable motif length (b) Variable sequence size (c) Variable frequency

Figure 8: Serial execution of ACME extracting exact length motifs using one core vs FLAME.

experiments, especially in cases where our competitors are too slow, we use only a prefix of these datasets. We executed our experiments on various architectures: (*i*) 32-bit Linux machine with 2 cores @2.16GHz and 2GB RAM, each core has 64KB L1 cache and 1MB L2 cache; (*ii*) 64-bit Linux machine with 12 cores @2.67GHz sharing 192GB RAM, each core has 64KB L1 cache and 256KB L2 cache with 12MB L3 cache; (*iii*) 64-bit Linux symmetric multiprocessing system (SMP) with 32 cores @2.27GHz sharing 624GB RAM, each core has 64KB L1 cache and 256KB L2 cache with 24MB L3 cache; (*iv*) IBM BlueGene/P supercomputer with 16,384 quad-core PowerPC processors @850MHz, with 4GB RAM per processor (64TB distributed RAM), each core has 64KB L1 cache and 2KB L2 cache with 8MB L3 cache.

6.1 CAST: Minimizing Cache Misses

Traverse is the process of going through the possible motif candidates. The available motif extraction methods pay a high cost in terms of cache misses during the *Traverse* phase. ACME, on the other hand uses our CAST approach to represent the search space in contiguous memory blocks. The goal of this experiment is to demonstrate the cache efficiency of CAST. We implemented the most common traversing mechanism utilized in the recent motif extraction methods, such as FLAME and MADMX, as discussed in Section 2. We refer to this common mechanism, as *NoCAST*.

We used the *perf* Linux profiling tool to measure the L1 cache misses. This test was done on the 2-core Linux machine. CAST significantly outperforms NoCAST in terms of cache misses and execution time especially when the motif length, and consequently the workload, is increasing, as shown in Figure 7.

6.2 Comparison Against State-of-the-art

We compared ACME to FLAME, MADMX, and VARUN based on different workloads. Since the source code for FLAME was not available, we implemented it using C++. MADMX[8] and VARUN[9] are available from their authors' web sites. These systems do not support parallel execution and are restricted to a particular motif type (exact-length or maximal motifs). The following experiments were executed on the 12-core Linux machine but, since our competitors run serially, for fairness ACME uses only one core. The reported time includes the suffix tree construction and mining time; the former is negligible compared to the mining time. Note

[8]http://www.dei.unipd.it/wdyn/?IDsezione=6376
[9]http://researcher.ibm.com/files/us-parida/varun.zip

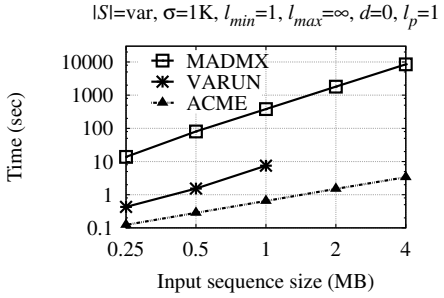
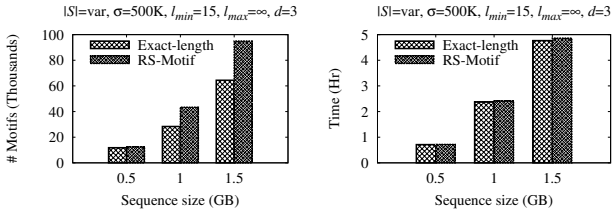


Figure 9: Serial execution of ACME extracting maximal motifs using one core vs MADMX and VARUN.



(a) Number of motifs (b) Time performance

Figure 10: Right-supermaximal vs Exact-length motifs extraction using ACME-THR in the 12-core machine.

that we use small datasets (i.e., up to 8MB from DNA), because our competitors cannot handle larger inputs.

FLAME and ACME produce identical exact-length motifs. The serial execution of ACME significantly outperforms FLAME with increasing workload, as illustrated in Figure 8. The impressive performance of ACME is a result of its cache efficiency. Note that if we were to allow ACME to utilize all cores, then it would be an order of magnitude faster than the serial version. For example, we tested the query of Figure 8(a) when motif length is 12: FLAME needs 4 hours, whereas parallel ACME finished in 7 minutes.

For maximal motifs, ACME is evaluated against MADMX and VARUN. Different similarity measures are utilized by ACME, MADMX and VARUN. Therefore, this experiment does not allow mismatches in order to produce the same results. Since the workload increases proportionally to the dis-

	Speedup Efficiencies w.r.t. $\mathcal{K}$ sub-tasks per core				
Cores	**$\leq$ 8**	**16–32**	**64–128**	**256–512**	**1,024 $\leq$**
512	0.85	0.94	0.99	0.97	0.81
1,024	0.73	0.87	0.97	0.97	0.83
2,048	0.83	0.92	0.97	0.83	
4,096	0.46	0.83	0.92	0.76	
8,192	0.25	0.76	0.46		

Table 2: Analysis to find a near-optimal value for $\mathcal{K}$ in Equation 1. ACME speedup efficiencies on Blue Gene/P using DNA with different prefix lengths. The Speedup is negatively affected by a small or very large $\mathcal{K}$.

tance threshold, this experiment is relatively of light workload. Again, for fairness all systems use only one core. Figure 9 shows that ACME is at least one order of magnitude faster than VARUN and two orders of magnitude faster than MADMX. Surprisingly, VARUN breaks while handling sequences longer than 1MB for this query, despite the fact that the machine has plenty of RAM (i.e., 192GB).

The next experiment demonstrates that ACME is capable of extracting rs-motifs from very long sequences with minimal overhead compared to maximal motifs. We use the 12-core machine and allow ACME to utilize all cores. Since we are executing only our method, we use 3 orders of magnitude larger datasets than the previous experiment (i.e., 0.5GB to 1.5GB from DNA). Figure 10(a) shows that there are significantly more rs-motifs compared to exact-length ones. Figure 10(b), however, shows that ACME needs only slightly more time to extract all rs-motifs, compared to the exact-length ones.

6.3 FAST: Optimizing Parallel Execution

In this section, we investigate the parallel scalability and speedup efficiency of ACME. We conducted strong scalability tests, where the number of cores increases while the problem size is fixed. The speedup efficiency measures the average utilization of $\mathcal{C}$ cores, and is calculated as $(\frac{\tau_1}{\mathcal{C}*\tau_\mathcal{C}})$, where τ_1 is the time of serial execution, and $\tau_\mathcal{C}$ is the time achieved using $\mathcal{C}$ cores. In the optimal case this ratio is 1.

We identify empirically a value for $\mathcal{K}$ in Equation 1 that allows ACME to achieve near-optimal speedup. Table 2 shows our analysis. Speedup efficiency is negatively affected by a small number of sub-tasks per core. This case appears when scaling to thousands of cores. For example, when $\mathcal{K} \leq 8$ the workload is imbalanced; thus the parallel performance is the lowest. Enlarging $\mathcal{K}$ is achieved by using a longer prefix. However, a longer prefix increases redundant work. Therefore, performance may be negatively affected, as shown in the case of $\mathcal{K} \in [64 - 128]$ and 8,192 cores.

We analyzed Equation 1 over different workloads. We empirically find $\mathcal{K}$=16 achieves a near-optimal speedup efficiency with different alphabet sizes and system architectures. Table 3(a) shows the results of a query over protein sequence on a Blue Gene/P supercomputer. Due to resource management restrictions, the minimum number of cores used in this experiment was 256 cores, and hence the speedup efficiencies are calculated relative to a 256-core system. With larger alphabets, Equation 1 leads to a small prefix length (l_p) yet the actual average number of sub-tasks is higher than 16. In Table 3(a), Equation 1 for 16,384 cores returns l_p=5; averaging $\frac{20^5}{16,384}$=195 sub-tasks per core. However, for 8,192 cores Equation 1 returns l_p=4; averaging 19 sub-tasks per core. This explains the better speedup efficiency with 16,384 compared to 8,192 cores.

$|S|$=32MB, σ=30K, l=12-∞, d=3

Cores	Hrs.	S.E.
256	19.83	1.00
1,024	4.97	0.99
2,048	2.51	0.98
4,096	1.29	0.96
8,192	0.68	0.91
16,384	0.31	0.98

(a) Protein; Blue Gene/P

$|S|$=1GB, σ=500K, l=12-∞, d=3

Cores	Hrs.	S.E.
1	15.95	1.00
3	6.30	0.84
6	4.23	0.63
12	2.63	0.51

(b) DNA; 12-core System

Table 3: Scalability of ACME on different computing architectures using different alphabets. S.E. denotes the speedup efficiency. ACME's S.E. for thousands of cores is strongly affected by the average number of tasks per core. Since ACME is self-tuned using Equation 1, the average number of tasks per core changes with the number of cores. Hence, S.E. does not necessarily decrease as the number of cores increases.

$|S| = Full\ Human\ Genome$, σ=500K, l_{min}=15, l_{max}=var, d=3

RS-Motifs ($l_{max} = \infty$)						Exact-length motifs ($l_{max} = l_{min}$)	
Len	Count	Len	Count	Len	Count	Length	Count
15	359,293	20	30,939	25	443	15	446,344
16	82,813	21	33,702	26	143	Total	446,344
17	22,314	22	12,793	27	37		
18	7,579	23	5,289	28	2		
19	2,288	24	2,435	Total	560,070		

Table 4: RS-Motifs from the complete human genome sequence (2.6GB) categorized by length. The total number of rs-motifs is more than total number of exact-length motifs.

Note that, although the query of Table 3(a) is against a sequence of size 32MB, it takes more than 10 days on an 8-core high-end workstation fully utilized by ACME. This is because the query workload is not only affected by sequence size but also by alphabet size, motif length, distance and frequency. ACME solved the same query in 18.6 minutes using 16,384 processors. The speedup of ACME on our 12-core shared memory machine for queries against DNA (1GB) is shown in Table 3(b). This experiment indicates that ACME is affected by the interference of using shared resources, such as memory and caches.

6.4 Interesting Findings in Real Datasets

In this section we demonstrate that ACME can provide useful insights into the properties of large real datasets, which are simply beyond the reach of any existing system. First we focus on the entire human genome (2.6GB). We use our 32-core machine and it takes around 10.5 hours to generate the results of Table 4. For example, if we allow distance

	Parameters	# Motifs	Longest	Time
DNA	σ=500K, l=12$-\infty$, d=2	5,937	20	0.6 min
Protein	σ=30K, l=12$-\infty$, d=1	96,806	95	2.1 min
English	σ=10K, l=12$-\infty$, d=1	315,732	42	3.5 min

Table 5: Analysis of three sequences of different alphabets, each of size 1GB, processed by ACME on a 12-core system.

$d = 3$, the longest motif that appears at least 500K times is 28 symbols long, and there are only two such motifs.

We also extracted 1GB long prefixes from the DNA, Protein and English datasets and ran queries with appropriate parameters for each dataset. For this experiment we used the smaller 12-core machine. The parameter settings and the results are summarized in Table 5. For example, if we allow distance $d = 1$ the longest motifs that appear in the English dataset (i.e., Wikipedia) at least 10,000 times are 42 characters long. Interestingly, these motifs are: "natural habitats are subtropical or tropical mo" and "united states the population was at the census en", possibly because the used Wikipedia extract was mainly geography-related.

7. CONCLUSION

Many important applications, such as bioinformatics, time series and log analysis, depend on motif extraction from one long sequence. Existing methods for extracting motifs from a single sequence are cache inefficient and serial. Parallelizing motif extraction attracted a lot of research efforts. However, most parallel motif extractors target a set of short sequences instead of a single long sequence.

This paper introduced ACME, a parallel combinatorial method for extracting motifs repeated in a single long sequence. ACME is based on two novel models, CAST and FAST to effectively utilize memory caches and processing power of multi-core shared-memory machines, and large-scale shared nothing systems with tens of thousands of processors, which are typical in cloud computing. ACME is 34 times faster than a recent exact-length motifs extractor, and 2 orders of magnitude faster than maximal motif extractors.

In our experiments we demonstrated that ACME handles the entire human genome on a single high-end multi-core machine; this is 3 orders of magnitude longer than what the state-of-the-art methods can support. Our system has practical applications in large-scale real life problems in bioinformatics, web log analysis, time series and other fields. Currently ACME is an in-memory system. We are working on a disk-based version that will allow ACME to support longer sequences in systems with limited memory.

8. REFERENCES

[1] A. Apostolico, M. Comin, and L. Parida. VARUN: discovering extensible motifs under saturation constraints. *IEEE/ACM Trans. on Comput. Biology Bioinformatics*, 7(4):752–26, 2010.

[2] V. Becher, A. Deymonnaz, and P. Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25(14):1746–53, 2009.

[3] A. M. Carvalho, A. L. Oliveira, A. T. Freitas, and M.-F. Sagot. A parallel algorithm for the extraction of structured motifs. In *Proc. of SAC*, pages 147–153, 2004.

[4] S. Challa and P. Thulasiraman. Protein sequence motif discovery on distributed supercomputer. In *Proc. of GPC*, pages 232–243, 2008.

[5] M. K. Das and H.-K. Dai. A survey of DNA motif finding algorithms. *BMC Bioinformatics*, 8(S-7), 2007.

[6] N. S. Dasari, R. Desh, et al. An efficient multicore implementation of planted motif problem. In *Proc. of HPCS*, pages 9–15, 2010.

[7] N. S. Dasari, D. Ranjan, and M. Zubair. High performance implementation of planted motif problem using suffix trees. In *Proc. of HPCS*, pages 200–206, 2011.

[8] M. Federico and N. Pisanti. Suffix tree characterization of maximal motifs in biological sequences. *Theoretical Computer Science*, 410(43):4391–4401, Oct. 2009.

[9] A. Floratou, S. Tata, and J. M. Patel. Efficient and Accurate Discovery of Patterns in Sequence Data Sets. *TKDE*, 23(8):1154–1168, Aug. 2011.

[10] R. Grossi, A. Pietracaprina, N. Pisanti, G. Pucci, E. Upfal, F. Vandin, S. Salzberg, and T. Warnow. MADMX: A Novel Strategy for Maximal Dense Motif Extraction. In *Proc. of Workshop on Algorithms in Bioinformatics*, pages 362–374, 2009.

[11] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of SIGMOD*, pages 1–12, 2000.

[13] C.-W. Huang, W.-S. Lee, and S.-Y. Hsieh. An improved heuristic algorithm for finding motif signals in DNA sequences. *IEEE/ACM Trans. on Comput. Biology Bioinformatics*, 8(4):959–975, 2011.

[14] Y. Liu, B. Schmidt, and D. L. Maskell. An ultrafast scalable many-core motif discovery algorithm for multiple gpus. In *Proc. of ISPA*, pages 428–434, 2011.

[15] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):1–41, 2010.

[16] A. Mueen and E. Keogh. Online discovery and maintenance of time series motifs. In *Proc. of ACM SIGKDD*, pages 1089–1098, 2010.

[17] M.-F. Sagot. Spelling Approximate Repeated or Common Motifs Using a Suffix Tree. In *Proc. of LATIN*, pages 374–390, Apr. 1998.

[18] K. Saxena and R. Shukla. Significant Interval and Frequent Pattern Discovery in Web Log Data. *Computing Research Repository (CoRR)*, abs/1002.1185, Feb. 2010.

[19] D. Tsirogiannis and N. Koudas. Suffix tree construction algorithms on modern hardware. In *Proc. of EDBT*, pages 263–274, 2010.

[20] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[21] X. Xie, T. S. Mikkelsen, A. Gnirke, K. Lindblad-Toh, M. Kellis, and E. S. Lander. Systematic discovery of regulatory motifs in conserved regions of the human genome, including thousands of ctcf insulator sites. *Proc. of the National Academy of Sciences*, 104(17):7145–7150, 2007.

[22] U. Yun and K. H. Ryu. Approximate weighted frequent pattern mining with/without noisy environments. *Knowledge-Based Systems*, 24(1):73–82, 2011.