The VLDB Journal manuscript No.
(will be inserted by the editor)

ACME: A Scalable Parallel System for Extracting Frequent Patterns from a Very Long Sequence

Majed Sahli · Essam Mansour · Panos Kalnis

Received: date / Accepted: date

Abstract Modern applications, including bioinformatics, time series, and web log analysis, require the extraction of frequent patterns, called motifs, from one very long (i.e., several gigabytes) sequence. Existing approaches are either heuristics that are error-prone; or exact (also called combinatorial) methods that are extremely slow, therefore applicable only to very small sequences (i.e., in the order of megabytes).

This paper presents ACME, a combinatorial approach that scales to gigabyte long sequences, and is the first to support supermaximal motifs. ACME is a versatile parallel system that can be deployed on desktop multi-core systems, or on thousands of CPUs in the cloud. However, merely using more compute nodes does not guarantee efficiency, because of the related overheads. To this end, ACME introduces an automatic tuning mechanism that suggests the appropriate number of CPUs to utilize, in order to meet the user constraints in terms of run time, while minimizing the financial cost of cloud resources. Our experiments show that, compared to the state of the art, ACME: supports 3 orders of magnitude longer sequences (e.g., DNA for the entire human genome); handles large alphabets (e.g., English alphabet for Wikipedia); scales out to 16,384 CPUs on a supercomputer; and supports elastic deployment in the cloud.

Majed Sahli · Panos Kalnis
King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
E-mail: majed.sahli@kaust.edu.sa; panos.kalnis@kaust.edu.sa

Essam Mansour
Qatar Computing Research Institute, Doha, Qatar
E-mail: emansour@qf.org.qa

1 Introduction

Applications such as human genome analysis in bioinformatics [30], stock market prediction in time series [22], and web log analytics [26], require the extraction of frequent patterns (i.e., *motifs*) from one very long sequence. This is known as the *repeated motifs* problem. Finding repeated motifs is computationally demanding and should not be confused with the much simpler common motifs problem [24], which focuses on a dataset of many short sequences. Repeated motif extraction approaches are classified into two categories: statistical and combinatorial [5]. The statistical ones rely on sampling or calculating the probability of motif existence, and trade accuracy for speed [14]; they may miss motifs (false negatives), or return motifs that do not exist (false positives). Combinatorial approaches [1,9,10], on the other hand, verify all combinations of symbols and return all motifs that satisfy the user's criteria. This paper focuses on the combinatorial case. Typically, data contain errors, noise and non-linear mappings [31]. Therefore, it is essential to allow occurrences of a motif to differ slightly according to a distance function.

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 $m =$ `GGTGC` be a candidate motif. Figure 1 shows subsequences 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 in Figure 1). 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, repeated motif extraction has three differences: (i) Order is important.

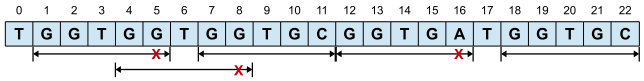


Fig. 1 Example sequence S over the DNA alphabet, Σ={A, C, G, T}. Five occurrences of motif candidate m = GGTGC are indicated, assuming distance threshold $d = 1$. X refers to a mismatch between m and the occurrence. Occurrences may overlap.

For example, AG may be frequent even if GA is infrequent. (*ii*) Motif occurrences may overlap. For example, in sequence AAA, the occurrences 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 σ=2 and d=1, TG is a valid 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 the length of the longest valid motif is l, the search space size is $\mathcal{O}(|\Sigma|^l)$.

To avoid the exponential increase in runtime, existing methods attempt to limit the search space by restricting the supported motif types [14,17]. FLAME [9], for instance, searches for motifs of a specific length only. Despite this restriction, the largest reported input sequence is only 1.3MB. Another way to limit the search space is by limiting the distance threshold. For example, MADMX [10] introduced the 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 does 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 the aforementioned methods are serial. To the best of our knowledge[1], there exists only one parallel approach, called PSmile [3]. It scales out to only 4 processors and the largest reported input is less than 0.25MB.

This paper presents ACME, a parallel system for motif extraction from a single long sequence. ACME is the first to support *supermaximal* motifs [8]; these are the longest motifs that are not subsequences of any other. Supermaximal motifs are very useful in practice because they provide a compact representation of the set of all motifs. We propose a novel algorithm that uses a trie to identify supermaximal motifs and avoids the huge overhead of storing all intermediate results. Additionally, ACME can extract exact-length and maximal motifs as defined in existing work.

ACME supports large scale parallelism. It partitions the search space into a large number (i.e., tens to hundreds of thousands) of independent tasks and employs a master-worker approach to facilitate *elasticity*. During runtime, ACME can dynamically scale in and out. Idle workers may leave and newly added ones request tasks from the master, keeping all CPUs busy. Note that, simply running on more CPUs does not guarantee the efficient utilization of resources. This is because over-partitioning of the search space may miss opportunities for early pruning, resulting in more and unnecessary work for many workers. In cloud computing environments, in particular, inefficient utilization of resources results in higher financial costs.

To maximize resource utilization, ACME implements a novel *automatic tuning* method. For each query, it generates a sample of representative tasks that are executed in order to collect statistics for the expected runtime. Then the system runs a set of simulations of a single-queue multiple-server model, for a varying number of tasks and CPUs. The outputs are a good decomposition of the search space, and an estimation of the expected runtime and speedup efficiency. The overhead of the auto-tuning process is minimal, but the benefits are significant: ACME achieves very good load balancing among CPUs, with almost perfect utilization in most cases. Moreover, given the pricing model of a cloud provider, auto-tuning suggests the optimal number of cloud computing resources (i.e., CPUs) to rent, in order to meet the user constraints in terms of runtime and financial cost. Our auto-tuning method is generic and applicable to a variety of architectures. We successfully deployed ACME on multi-core shared memory workstations; shared-nothing Linux clusters (both local and in Amazon EC2); and a large supercomputer with 16,384 CPUs.

ACME scales to gigabyte-long sequences, such as the DNA for the entire human genome (2.6GBps). Similar to some existing methods, we use a suffix tree [11] to keep occurrence counts for all suffixes in the input sequence. 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. Both are arranged in a way that exhibits spatial and temporal locality. This allows us to store the data in contiguous memory blocks that are kept in the CPU caches, and minimize cache misses in modern architectures. By being cache-efficient, ACME achieves almost an order of magnitude performance improvement for *serial* execution.

In summary, our contributions are:

[1] There exist several parallel approaches [4,6,7,16] for the much simpler common motifs problem.

– We propose a parallel approach that decomposes the motif extraction process into fine-grained tasks, allowing for the efficient utilization of thousands of processors. ACME scales to 16,384 processors on an IBM Blue Gene/P supercomputer and solves in 18 minutes a query that needs more than 10 days on a high-end multi-core machine.
– We develop an automatic tuning method that facilitates near optimal utilization of resources, and is especially useful for cloud environments.
– We are the first to support supermaximal motifs with minimal overhead.
– We develop a cache-efficient search space traversal technique that improves the serial execution time by almost an order of magnitude.
– We conduct experimental evaluation with large real datasets on different architectures, locally and on the cloud. ACME scales to large alphabets (e.g., English alphabet for the Wikipedia dataset) and handles 3 orders of magnitude longer sequences than our competitors on the same machine. We are the first to support gigabyte long sequences, such as the entire human genome.

The rest of this paper is organized as follows. Sections 2 and 3 present the background and related work. Section 4 presents our algorithm for supermaximal motifs. Section 5 contains the details of our parallel approach. Automatic tuning and elasticity are discussed in Section 6, whereas Section 7 focuses on our cache-efficient implementation. Section 8 presents the experimental evaluation and Section 9 concludes the paper.

2 Background

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: $\mathcal{L}(m) = \{(i, j) | \mathcal{D}(S[i, j], m) \leq d\}$.

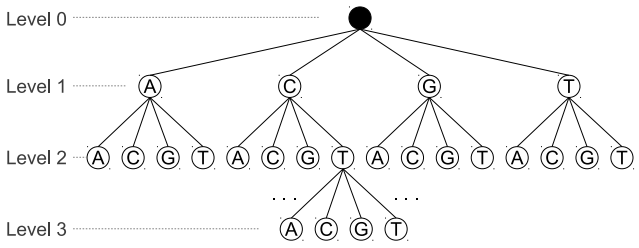


Fig. 2 Partial three levels of the combinatorial search space trie for DNA motifs, alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

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: $|\mathcal{L}(m)| \geq \sigma$.*

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.*

A maximal motif must be right- and left-maximal [8]. m is right-maximal if $\mathcal{L}(m\alpha)$ has less occurrences or more mismatches than $\mathcal{L}(m)$, where $\alpha \in \Sigma$. Similarly, a motif m is left-maximal if extending m to the left results in fewer occurrences or introduces new mismatches. There exists excessive overlapping among maximal motifs, with lots of short motifs contained in longer ones.

Definition 3 (Supermaximal Motif) *Let M be the set of maximal motifs from Definition 2 and let $\widehat{m} \in M$. $\widehat{m}$ is a* supermaximal *motif, if $\widehat{m}$ is not a subsequence of any other motif in M. The set of all supermaximal motifs is denoted by M_s.*

The number of possible motif candidates is $\mathcal{O}(|\Sigma|^l)$, where $|\Sigma|$ is the alphabet size and l is the length of the longest candidate; for a certain σ value, the number of candidates is upper-bounded by $\sum_{i=1}^{|S|-\sigma+1} |\Sigma|^i$. To restrict the number of candidates, previous works have imposed minimum (l_{min}) and maximum (l_{max}) length constraints. The most interesting case is when $l_{max} = \infty$. Obviously, this is also the most computationally expensive case since length cannot be used for pruning. This paper solves efficiently the following problem (including the case where $l_{max} = \infty$):

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 supermaximal motifs.*

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; as mentioned before, the search space grows exponentially to the length

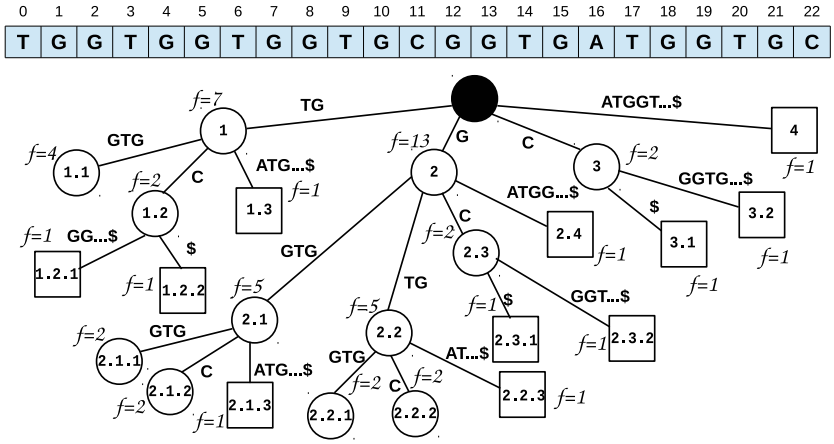


Fig. 3 Example sequence over $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and its suffix tree. \$ is the termination symbol. Squares denote leaves. Nodes are numbered for referencing and annotated with the frequency of their path labels (number of reachable leaves). Only part of the tree is shown; the entire tree would have 24 leaves (i.e., one for each of the 23 suffixes plus one for the termination symbol).

of the longest candidate. 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 [11] is typically used.

A suffix tree is a full-text index that groups identical subsequences. Let q be a query string. Using a suffix tree, we can check if q appears in S in time linear to the length of q. Figure 3 shows an example suffix tree. First the input string S is appended with a termination symbol \$. Then all suffixes $S[0], S[1], \ldots$ of S are generated. In our example, $S[11] = \mathtt{CGG...\$}$ and $S[22] = \mathtt{C\$}$ correspond to the suffixes that start at positions 11 and 22, respectively. All suffixes are inserted in a Patricia trie, such that a path label from the root to a leaf corresponds to a suffix. For instance, the path labels from the root to leaves `3.1` and `3.2` correspond to suffixes $S[22]$ and $S[11]$, respectively. Observe that the path label from the root to internal node `3` is `C`, which is the common prefix of $S[22]$ and $S[11]$. The suffix tree is built in linear time and space as long as S and the tree fit in memory [29]. There are also efficient and parallel suffix tree construction algorithms [18] for longer sequences.

The suffix tree can be used to verify efficiently if a motif candidate is a maximal motif. Recall from Section 2.1 that a maximal motif must be left- and right-maximal. Federico and Pisanti [8] show how to check these properties using the suffix tree: (*i*) A motif m is *left-maximal* if one of its occurrences corresponds to a *left-diverse* suffix tree node. A suffix tree node is left-diverse if at least two of its descendant leaves have different preceding symbols in S. For example (see Figure 3), nodes `2.1.3` and `2.4` correspond to $S[12] = \mathtt{GGTGA...\$}$ and $S[15] = \mathtt{GAT...\$}$, respectively. The preceding symbols in S are $S[11] = \mathtt{C}$ and $S[14] = \mathtt{T}$; that is, they are different. Both `2.1.3` and `2.4` are leaves under node `2`; therefore, node `2` is left-diverse. As a counter-example, consider node `1.2`: its path label, `TGC`, is always preceded by `G` in S; therefore it is not left-diverse. (*ii*) By construction, 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, it is right maximal. For example, motif $m = \mathtt{GTG}$ consumes the entire path label from the root to node `2.2`. The labels to the children of node `2.2` start with three different symbols: `G`, `C` and `A`. Consequently, m cannot be extended to the right without changing its set of occurrences, so by definition m is right-maximal.

We annotate the suffix tree by traversing it once and storing in every node whether it is left-diverse, and the number of leaves reachable through it. This number is the frequency of the node's path label. For example, node `1.2` in Figure 3 is annotated with $f = 2$ because its path label `TGC` appears in S at $(9, 11)$ and $(20, 22)$. For clarity, we do not show the left-diversity annotation in the figure. For the special case of exact motifs, where the distance threshold $d = 0$, 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 candidate is the sum of the frequencies of all these nodes.

Example. Assume $d = 1$, $\sigma = 10$ and start a depth-first traversal (DFT) of the search space in Figure 2. The first candidate is $m = \mathtt{A}$. Traverse the suffix tree in Figure 3 and note that the first symbol from every

Table 1 Comparison of combinatorial motif extractors for the *repeated* motif problem.

				Supported motif types		
	Index	Parallel	Largest reported input	Exact-length	Maximal	Supermaximal
FLAME [9]	Suffix Tree		1.3 MB	✓		
VARUN [1]	N/A		3.1 MB		✓	
MADMX [10]	Suffix Tree		0.5 MB		✓	
PSmile [3]	Suffix Tree	✓	0.2 MB	✓		
ACME [our's]	Suffix Tree	✓	**2.6 GB**	✓	✓	✓

branch starting at the root differs from A by at most $1 \leq d$. Therefore, the occurrences set contains the following suffix tree nodes: $\mathcal{L}(\texttt{A}) = \{1, 2, 3, 4\}$ with total frequency: 7+13+2+1 = 23 $\geq \sigma$. Continue the DFT in Figure 2 and generate a new candidate motif $m' = \texttt{AA}$. Search the suffix tree starting from the nodes in $\mathcal{L}(\texttt{A})$. The path label of suffix tree node 1 is TG; its distance from AA is $2 > d$, so it is discarded. Next, check all branches of suffix tree node 2. Its first three children, 2.1, 2.2, 2.3, are discarded for exceeding the allowed distance. Child 2.4 is added to the occurrences set of AA since its path label is GA, which has distance $1 \leq d$ from AA. The rest of the nodes in $\mathcal{L}(\texttt{A})$ are extended and validated in the same manner. The resulting occurrences set of m' is $\mathcal{L}(\texttt{AA}) = \{2.4, 4\}$ with total frequency $1 + 1 = 2 < \sigma$. m' is not frequent enough, so the search space is pruned by backtracking to node A in Figure 2. Then, DFT generates candidate $m'' = \texttt{AC}$, which is checked in the same way. The process continues until the search space of Figure 2 is exhausted. □

3 Related Work

This section presents the most recent methods for extracting motifs from a single sequence (i.e., combinatorial repeated motifs problem); Table 1 shows a summary. 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 [17].

The complexity of motif extraction grows exponentially with the motif length. Intuitively, extracting maximal and supermaximal motifs is more complex than exact-length ones because, if length is known, the search space can be pruned significantly. FLAME [9] supports only exact-length motifs. To explore a sequence, users need to run multiple exact-length queries. VARUN [1] and MADMX [10], on the other hand, support maximal motifs, without any length restriction. To limit the search space, VARUN and MADMX assume that the distance threshold varies with respect to the length of the current motif candidate. None of these techniques supports supermaximal motifs; therefore their output contains a lot of redundant motifs (i.e., motifs that are subsequences of longer ones). Despite these restrictions, the length of the largest reported input was only a few megabytes. It must be mentioned that none of these methods is parallel.

Parallelizing motif extraction attracted a lot of research efforts, especially in bioinformatics [3,4,6,7,16, 19]. Challa and Thulasiraman [4] handle a dataset of 15,000 protein sequences with the longest sequence being 577 symbols only; their method does not manage to scale to more than 64 cores. Dasari et al. [6] extract common motifs from 20 sequences of a total size of 12KB and scale to 16 cores. This work has been extended [7] to support GPUs and scaled to 4 GPU devices using the same dataset. Liu et al. [16] process a 1MB dataset on 8 GPUs. DMF, an implementation of Huang's work [13], has been parallelized by Marchand et al. [19] to run on a supercomputer. The aforementioned methods target the much simpler *common* motifs problem (i.e., they assume a dataset of many short sequences), whereas we solve the *repeated* motifs problem [24] (i.e., one very long sequence). Moreover, most of these approaches are statistical. Therefore they may introduce false positive or false negative results, whereas we focus on the combinatorial case that guarantees correctness and completeness.

To the best of our knowledge, the only parallel and combinatorial method for extracting motifs from a single sequence is PSmile [3]. This method parallelizes SMILE [20], an existing serial algorithm that extracts structured motifs, composed of several "boxes" separated by "spacers" of different lengths. Intuitively, this corresponds to a distance function that allows gaps. SMILE (and PSmile) can support the Hamming distance by using only one box and zero-length spacers. Similar to our approach, PSmile partitions the search space using prefixes. Their contribution is the heuristic that groups search space partitions to balance workload among CPUs. Prefixes are grouped into coarse-grained tasks, and workload of different prefixes is assumed to be similar. Based on this assumption, a static scheduling scheme is used to distribute the tasks. In the follow-

ing, we will explain that their assumption is not satisfied in real datasets; therefore, in practice PSmile suffers from highly imbalanced workload and parallel overhead [28]. PSmile reported scaling to 4 CPUs only; the maximum input size was 0.2MB. In contrast, our approach scales to 16,386 CPUs and can support gigabyte long sequences.

In our recent conference paper [25] we introduced CAST, a cache-aware method for solving the combinatorial repeated motifs problem. CAST arranges the suffix tree in continuous memory blocks, and accesses it in a way that exhibits spatial and temporal locality, thus minimizing cache misses. As a result, CAST improves performance of serial execution by an order of magnitude. Our conference paper also supports *right*-supermaximal motifs, which are motifs that are not prefixes of any other. Right-supermaximal motifs remove some of the redundant maximal motifs that were reported by previous work. Finally, our conference paper introduced the first parallel method to scale to thousands of CPUs and gigabyte long sequences. The most important issue for good scalability is to achieve load balance by having a good decomposition of the search space. In our conference paper, we used a naïve trial-and-error approach to find a good decomposition for each query in an ad-hoc manner.

This work extends our conference paper in two ways: (*i*) We support *supermaximal* motifs. Straight-forward calculation would require keeping track of all maximal ones; this is too expensive, therefore no previous approach supports such motifs. We propose an algorithm to extract supermaximal motifs with minimal overhead. (*ii*) We develop an automatic tuning process for the partitioning of the search space in order to scale efficiently to thousands of CPUs. For each query and dataset, we gather statistics and build an execution model. We then run simulations to decide the best partitioning that maximizes the efficient utilization of available CPUs. In conjunction with a cloud provider's pricing scheme, our auto-tuning method can also be used to minimize the financial cost of deployment on the cloud, while meeting the user constraints in terms of performance.

4 Supermaximal Motifs

Supermaximal motifs are those that are not contained in any other motif. They are very useful in practice, since they provide a compact and comprehensive representation of the set of all motifs. However, naïve methods that compute supermaximal motifs require to maintain the complete set of maximal ones [8]. The set of maximal motifs is prohibitively large for typical inputs and queries, and the verification process is computationally very expensive. For this reason, none of the existing systems supports supermaximal motifs.

Input: Empty trie
Output: Supermaximal motifs

```
1 while Workers Exist do
2     buffer ← ReceiveFromWorker()
3     foreach motif in buffer do
4         reversed ← Reverse(motif)
5         InsertInTrie(reversed)
6 SpellTrieFromLeaves()
```

Algorithm 1: Supermaximal Motifs

Below, we propose a novel algorithm for extracting supermaximal motifs without storing the complete set of intermediate results. In the experimental section we will show that our algorithm poses minimal overhead, compared to existing methods that only find maximal motifs. Our solution is based on the following observations:

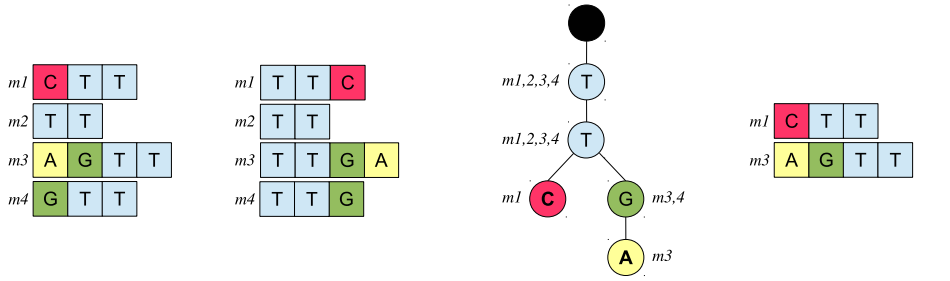
Observation 1 *Let $\alpha m \beta$ be a supermaximal motif. The set of maximal motifs M may contain motifs $\{\alpha,\ \alpha m,\ \alpha m\beta,\ m,\ m\beta,\ \beta\}$.*

Observation 2 *The set of supermaximal motifs M_s does not contain prefixes $M_{pre} = \{\alpha, \alpha m\}$, or suffixes $M_{suf} = \{m\beta, \beta\}$, or subsequences $M_{sub} = \{m\}$.*

Example. Let the set of maximal motifs for a certain query be $M = \{\texttt{AGTT}, \texttt{GTT}, \texttt{TT}, \texttt{AGT}, \texttt{AG}, \texttt{GT}, \texttt{CTT}, \texttt{CT}\}$. To find the set of supermaximal motifs M_s, we have to eliminate maximal motifs that are subsequences of other ones. According to our observations, (*i*) $M_{pre} = \{\texttt{AGT}, \texttt{AG}, \texttt{CT}\}$, (*ii*) $M_{suf} = \{\texttt{GTT}, \texttt{TT}\}$, and (*iii*) $M_{sub} = \{\texttt{CT}, \texttt{GT}\}$. Therefore, $M_s = \{\texttt{AGTT}, \texttt{CTT}\}$. □

During the depth-first traversal of the search space, motifs that share the same prefix are grouped in the same sub-trie (see Figure 2); hence, we are able to easily filter motifs that are prefixes of other ones. The longest valid branches represent the set of maximal motifs that do not belong to M_{pre} or M_{sub}. We refer to this set as the right-supermaximal [25] set $M_{rs} = M_s \cup M_{suf}$. In our example, $M_{rs} = \{\texttt{AGTT}, \texttt{CTT}, \texttt{GTT}, \texttt{TT}\}$. Now, we can find the supermaximal motifs by discarding all proper suffixes from M_{rs}. However, computing M_{suf} is challenging, because motifs in M_{rs} belong to different parts of the search space as they start with different prefixes. A naïve solution would check all possible pairs in M_{rs}; the complexity of such a solution is $\mathcal{O}(|M_{rs}|^2)$.

We propose Algorithm 1, which in the average case removes redundant suffixes in $\mathcal{O}(|M_{rs}| \log_{|\Sigma|} |M_{rs}|)$ time.



(a) The M_{rs} set (b) Motif in M_{rs} reversed (c) Trie of reversed motifs (d) Supermaximal motifs

Fig. 4 The steps for extracting the set of supermaximal motifs M_s from the set of intermediate results M_{rs}.

The algorithm reverses the contents of M_{rs}, effectively transforming the problem from suffix- to prefix-removal. As we mentioned earlier, the latter can be solved efficiently by utilizing a trie.

Figure 4(a) depicts the M_{rs} set for our running example; the conceptually reversed motifs are shown in Figure 4(b). The reversed motifs are inserted in a trie that is shown in Figure 4(c); observe that common prefixes are grouped together. In the trie, each path from the root to a leaf corresponds to a string that is not a prefix of any other. Our example trie has two leaves. After reversing back the corresponding paths, the final set $M_s = \{\texttt{AGTT}, \texttt{CTT}\}$ of supermaximal motifs is shown in Figure 4(d).

The set of supermaximal motifs from Algorithm 1 is correct and complete. Refer to Appendix A for the proof.

5 Parallel Motif Extraction (FAST)

This section presents FAST[2], our efficient parallel space traversal approach that scales to thousands of CPUs[3]. FAST achieves high degree of concurrency by partitioning the search space horizontally and balancing the workload among CPUs with minimal communication overhead.

5.1 System Architecture

We adopt the master-worker architecture shown in Figure 5. Given $\mathcal{C}$ CPUs, one master generates tasks; $\mathcal{C}-2$ workers request tasks from the master and generate right-supermaximal motifs; and one combiner receives the intermediate results from the workers and extracts supermaximal motifs. The details are explained below.

[2] FAST stands for Fine-grained Adaptive Sub-Tasks

[3] For simplicity, the discussion assumes that each CPU executes a single process. In practice, our implementation executes one process per *core*.

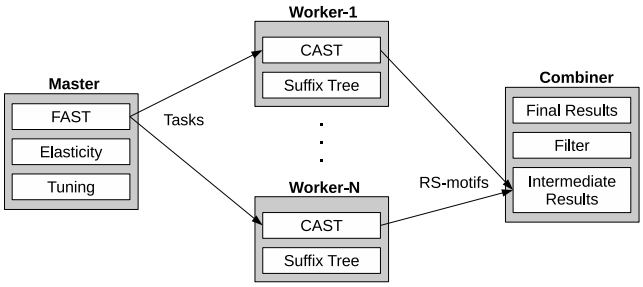


Fig. 5 Architecture of parallel ACME.

Master. First the master reads the sequence from the disk and sends it to the workers. Then it decomposes the search space and starts generating tasks. The decomposition of the search space is based on our automatic tuning model, presented in Section 6; its goal is to utilize efficiently the available CPUs. Given a decomposition, the master uses our FAST technique, discussed in Section 5.4, to generate tasks. Tasks are scheduled dynamically using pull requests from workers.

Worker. Each worker receives the input sequence and constructs the annotated suffix tree (see Figure 3). Every worker needs access to the entire suffix tree, because occurrences of a motif candidate can occur at different branches. Once the index is ready, the worker requests a task from the master. Tasks are processed using our CAST technique to find right-supermaximal motifs; refer to Section 7 for details. The right-supermaximal motifs from each task are sent to the combiner in batches. Results within a batch share the same prefix; therefore, the prefix is stripped to better utilize the limited buffer space and minimize communication cost. When a worker is free it requests a new task from the master. This simple scheduling scheme allows workers to enter or leave the system anytime.

Combiner. The combiner implements Algorithm 1: it receives right-supermaximal motifs from all workers and uses a trie (see Figure 4(c)) to generate the final result, that is the set of supermaximal motifs. We will show in the experimental evaluation that the workload of the

combiner is minimal, compared to the workers'. Therefore, the combiner is not a bottleneck and does not affect the scalability of the system.

5.2 Horizontal Search Space Partitioning

The search space depicted in Figure 2 can be split into independent sub-tries. Parallelizing the trie traversal is easy in this sense. However, the motif extraction search space is pruned at different levels 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 affects the efficiency of parallel systems due to underutilized resources.

FAST decomposes the search space into a large number of independent sub-tries. Our target is to provide enough sub-tries per CPU to utilize all computing resources with minimal idle time. We partition horizontally the search space at a certain depth l_p, into a fixed-depth sub-trie and a set of variable-depth sub-tries, as shown in Figure 6; observe that l_p corresponds to the *prefix length*. Since the search space is a combinatorial trie, there are $|\Sigma|^{l_p}$ sub-tries. The variable-depth sub-tries are of arbitrary size and shape due to the pruning of motif candidates at different levels.

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} 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 4^2=16 large variable-depth sub-tries. Each sub-trie consists of 4^{13} branches (more than 67 million). If the horizontal partition cuts at depth 8, then there are 4^8=65,536 independent and small variable-depth sub-tries with 16 thousand branches each. □

5.3 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 example, let $|\Sigma| = 4$ and let `AA` be a prefix that leads to no valid motifs. Using a prefix length of 5 (i.e., horizontal partitioning at depth 5) introduces 64 false positive prefixes that start with `AA`. Therefore, although the longer prefix increases the number of tasks (i.e., increases the degree of concurrency), the resulting false positive prefixes introduce useless overhead and, consequently, suboptimal utilization of resources.

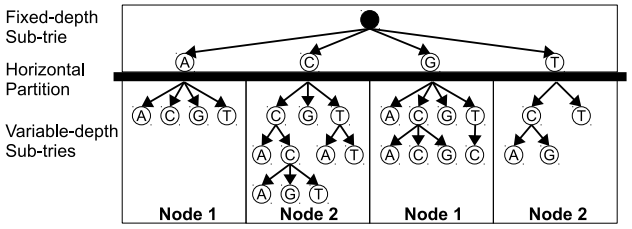


Fig. 6 Combinatorial trie partitioned at depth $l_p = 1$ into a fixed-depth sub-trie leading to four variable-depth sub-tries, which are traversed simultaneously by two compute nodes.

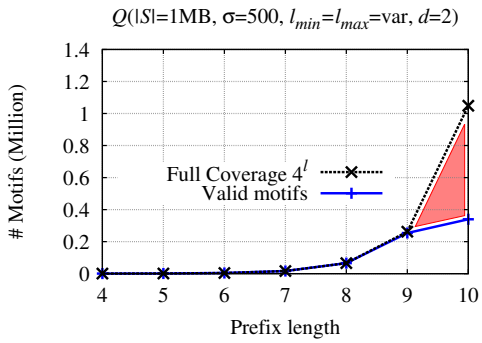
Observation 3 *Given distance threshold d, all prefixes of length d are valid (i.e., cannot be pruned earlier).*

Let S be the input sequence. Any subsequence of S of length l will not exceed the distance threshold d for all search space 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 3 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. Figure 7(a) shows an experiment where prefixes of length up to 9 symbols are fully covered although the sequence size is only 1MB. In this experiment, the prefix of length 10 leads to more than 0.5M false positive prefixes, that is, useless tasks that will be processed.

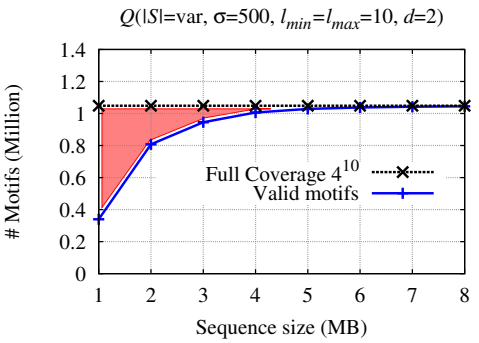
Observation 4 *As the input sequence size increases, the depth of the search space with full coverage increases.*

A longer sequence over a certain alphabet Σ means more repetitions of subsequences. 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 7(b) shows an experiment where the number of false positive prefixes generated at $l_p = 10$ in the 1MB sequence decreases by increasing the sequence size.

Observation 5 *If the search space is horizontally partitioned at depth l_p, where the average number of sub-tries per CPU leads to high resource utilization, then a longer prefix is not desirable to avoid the overhead of false positives.*



(a) Variable motif length



(b) Variable sequence size

Fig. 7 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.

5.4 FAST Algorithm

FAST generates enough independent tasks per CPU, to maximize the utilization of CPUs. A task consists of one or more sub-tries and is transferred in a fixed-length compact form. The master horizontally partitions the search space and schedules tasks as shown in Algorithm 2. Function GetOptimalLength in line 1 calculates the near-optimal prefix length that will achieve the best load balance; the details will be explained in Section 6. The exact-length prefixes are generated by 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 significant speedup.

Input: Alphabet Σ, Number of CPUs $\mathcal{C}$, Task size factor λ
Output: Generate and schedule tasks

```
   // Calculate optimal prefix length
1  l_p ← GetOptimalLength(Σ,C)
2  i ← 0 // An iterator over all prefixes
   // Calculate task size
3  t ← ⌊λ|Σ|^{l_p}/C⌋ // one task contains t prefixes
   // Assign tasks
4  while i ≠ prefixes end do
5  |  task ← GetNextPrefix(i, t)
6  |  WaitForWorkRequest( )
7  |  SendToRequester(task)
8  |  i ← i + t
   // Signal workers to end
9  while worker exist do
10 |  WaitForWorkRequest( )
11 |  SendToRequester(end)
```

Algorithm 2: Partitioning and Scheduling

6 Automatic Tuning and Elasticity

This section discusses the automatic tuning feature that allows ACME to utilize efficiently thousands of CPUs. The section also discusses our elasticity model that suggests the appropriate amount of cloud resources to rent while meeting the user's requirements in terms of processing time and financial cost.

6.1 Problem Definition

The goal of automatic tuning is to find a good decomposition of the search space (i.e., parameter l_p) that minimizes runtime, while achieving high utilization of computational resources. To minimize runtime, we need to utilize efficiently as many CPUs as possible, which translates to: (*i*) enough tasks per CPU, in order to achieve good load balance; and (*ii*) few false positives, in order to avoid useless work. As explained in the previous section these goals contradict each other. Therefore, we need to solve the following optimization problem:

Problem 2 *Find the value of parameter l_p that maximizes scalability (i.e., number of CPUs) under the constraint that speedup efficiency $SE \geq SE_{min}$.*

Let $\mathcal{C}$ be the number of workers, T_1 the time to execute the query using one worker (i.e., serial execution) and $T_{\mathcal{C}}$ the time to execute the query using $\mathcal{C}$ workers. *Speedup efficiency* is defined as:

$$SE = \frac{T_1}{\mathcal{C} \cdot T_{\mathcal{C}}} \quad (1)$$

The maximum value for SE is 1, indicating perfect parallelism. In practice there are always overheads, there-

Table 2 Example query Q running on 240 workers. For $l_p = 2$, we cannot achieve load balance. For $l_p = 4$, there are too many false positive tasks. The optimal search space decomposition is found using $l_p = 3$, achieving very good speedup efficiency $SE = 0.91$.

$Q(|S|$=32MB, σ=30K, l_{min}=7, l_{max}=∞, d=2)

Prefix Length (l_p)	Tasks ($\|\Sigma\|^{l_p}$)	Average Tasks/Worker	Speedup Efficiency
2	400	1.66	0.47
3	8,000	33.33	0.91
4	160,000	666.66	0.22

fore we require $SE \geq SE_{min}$, where SE_{min} is a user-defined threshold. Typically, $SE_{min} = 0.8$ is considered good in practice.

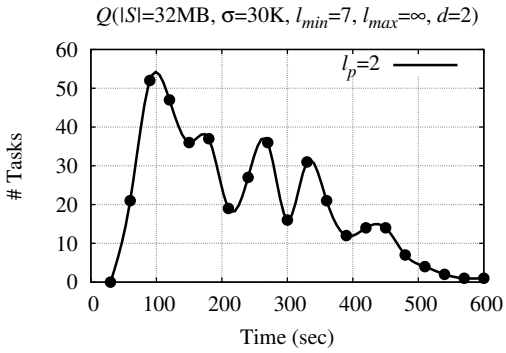
Example. Let us consider a query Q that searches a 32MB protein sequence (i.e., alphabet size $|\Sigma| = 20$) for supermaximal motifs that appear at least $\sigma = 30{,}000$ times with distance threshold $d = 2$; the minimum length should be $l_{min} = 7$ and there is no maximum length limit (i.e., $l_{max} = \infty$). We generated different decompositions of the search space using $l_p = 2$, 3, 4, executed the query on 240 workers, and measured the run time. The resulting values for SE are shown in Table 2. When $l_p = 2$, on average there are only 1.66 tasks per worker, so it is difficult to achieve load balance; consequently SE is only 0.47. For $l_p = 4$, on the other hand, there are a lot of false positive tasks, resulting in very low speedup efficiency (i.e., only 0.22). For this particular query, the optimal space decomposition is reached for $l_p = 3$; achieving $SE = 0.91$, which is very good in practice. □

Since the processing time of each task is not known in advance, it is difficult to find an analytical solution for Problem 2; therefore our solution is based on heuristics. Note that the accuracy of the results is *not* affected; our algorithm will still return the correct and complete set of supermaximal motifs. If our heuristics fail to achieve optimal space decomposition, then only the execution time will be affected, due to sub-optimal utilization of computational resources.

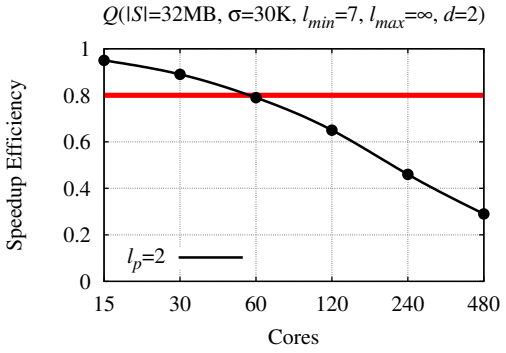
6.2 Distribution of Workload Frequency

In the following, we will explain the results of Table 2 by analyzing the workload frequency distribution of the tasks and its effect on scalability, for different search space decompositions. We will reuse the same example query Q from the previous section.

Let us start with prefix length $l_p = 2$ that decomposes the search space of Q into $20^2 = 400$ tasks (recall that the alphabet contains 20 symbols). We run each task on one CPU and measure its execution time. The



(a) Workload frequency distribution for 400 tasks

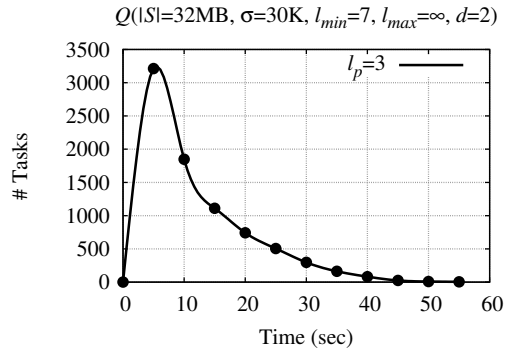


(b) Speedup efficiency as number of cores is varied

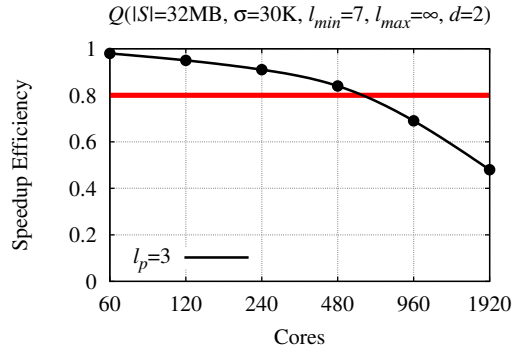
Fig. 8 For example query Q and $l_p = 2$, the search space is decomposed to 400 large tasks. Load balancing is poor and the speedup efficiency drops when using more than 60 CPUs.

results are shown in Figure 8(a), which represents the *workload frequency distribution* for the combination of Q and l_p. For a point (x, y), x represents execution time, whereas y represents the number of tasks that require time x to run. The total execution time for Q is given by the area under the curve.

The coarse decomposition of the search space leads to an irregular distribution with many "heavy" tasks. For instance, there are about 70 tasks that run in less than 100sec, but there are also around 130 tasks that need more than 300sec; some extreme cases need more than 500sec. Even with dynamic scheduling, balancing such a workload on a parallel system is challenging. We executed Q with varying number of CPUs and measured the speedup efficiency SE; the results are shown in Figure 8(b). Assuming the threshold for good speedup efficiency is $SE_{min} = 0.8$, the figure shows that this particular decomposition does not allow Q to scale efficiently to more than 60 CPUs. Note that, if more than 60 CPUs are used the total execution time for Q will decrease, but due to load imbalance many CPUs will be underutilized, so computational resources will be wasted. In our experiment, if instead of 60 we use

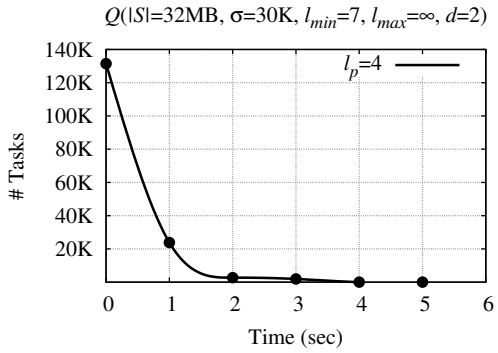


(a) Workload frequency distribution for 8,000 tasks

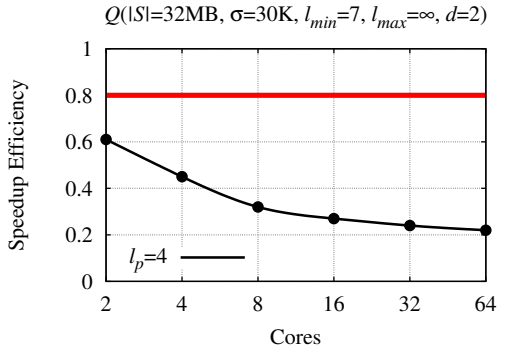


(b) Speedup efficiency as number of cores is varied

Fig. 9 For example query Q and $l_p = 3$, the search space is decomposed to 8,000 tasks. Load balancing is near optimal and the speedup efficiency is high up to 500 CPUs.



(a) Workload frequency distribution for 160K tasks



(b) Speedup efficiency as number of cores is varied

Fig. 10 For example query Q and $l_p = 4$, the search space is decomposed to 160,000 tasks. Since most tasks are false positive prefixes, speedup efficiency is poor even at 2 CPUs.

480 CPUs (i.e., 8x increase), the total execution time will drop from 30min to 10min (i.e., only 3x improvement). This is the practical meaning of low SE values.

Our scheduling corresponds to an instance of the online dynamic bin packing problem. When items are few and large (i.e., coarse decomposition of search space), bins cannot be filled optimally. Intuitively, more and smaller objects are needed. This corresponds to a longer prefix length, resulting in a finer decomposition. We run again the same experiments for $l_p = 3$, which generates $20^3 = 8,000$ tasks. The workload frequency distribution is shown in Figure 9(a); it resembles a leptokurtic and positively skewed non-symmetric distribution. While we do not know the processing time of tasks beforehand, we expect their execution time to decrease monotonically as they are further decomposed. Indeed, the figure shows that the majority of tasks run in around 5sec, whereas very few need from 40 to 60sec. Consequently, there are enough small tasks to keep all CPUs busy while the few larger ones are executed. Moreover, the probability of a large task being executed last is low because there are only a few of them; therefore, we expect good load balance. Figure 9(b) shows the speedup efficiency for a varying number of CPUs. Observe that the algorithm scales well (i.e., $SE \geq 0.8$) up to about 500 CPUs, which is an order of magnitude more, compared to Figure 8(b).

It is tempting to generate an even finer search space decomposition in order to scale to more CPUs. Figure 10(a) shows the workload frequency distribution for $l_p = 4$. The graph resembles a power-law distribution. Out of the 160,000 generated tasks, very few take 3 to 5sec, whereas the vast majority (i.e., around 130,000 tasks) are very small with execution time close to zero. Unfortunately, many of these tasks are false positives that generate useless work. Although the overhead per false positive task is small, because of their sheer number, the cumulative overhead is high. Figure 10(b) shows the speedup efficiency for a varying number of CPUs. SE is always less than SE_{min}; therefore, for this decomposition the system cannot scale efficiently not even on 2 CPUs.

6.3 ACME Automatic Tuning

We solve Problem 2 as follows: We partition the search space at a specific prefix length l_p and draw a random sample of tasks to run, in order to estimate the speedup efficiency SE. We repeat this process with different prefix lengths until we find the one that allows scaling to the largest number of CPUs with $SE \geq SE_{min}$.

Algorithm 3 describes the process. In line 1, l_p is initialized to the minimum motif length l_{min}. We start with the longest prefix length possible, that is the minimum length of valid motifs; then we try shorter prefixes. This way we arrive to the optimal l_p faster, because longer prefixes produce smaller tasks that run faster. If l_p is decremented to the distance value d without meeting the stopping criterion (see line 7), l_p is set to $d+1$ and the algorithm terminates. This follows from Observation 1 in Section 5.3. To reduce the overhead of the automatic tuning process, sample prefixes can be generated and evaluated in parallel (i.e., lines 4 and 5). In practice, the main loop of the algorithm is executed only a few times before finding a near-optimal decomposition.

Function EstSpdupEff in line 6 is the heart of the algorithm. Given a decomposition, for a specific number $\mathcal{C}$ of CPUs, it estimates the corresponding speedup efficiency. The function iterates over a range of values for $\mathcal{C}$ and returns the one that achieves the maximum SE for the given space decomposition. The following paragraphs explain how to estimate the serial (i.e., T_1) and parallel (i.e., $T_{\mathcal{C}}$) execution times, which are required by EstSpdupEff.

Input: Sequence S; query $Q(|S|, \sigma, l_{min}, l_{max}, d)$; threshold SE_{min}
Output: Prefix length l_p; number of CPUs $\mathcal{C}_{max}$

```
1  l_p ← l_min
2  C_max ← 1
3  while l_p > d do
       // randomly draw x prefixes of length l_p
4      sample ← RandomPrefixes(x, l_p)
5      sample_times ← ExtractMotifs(sample)
6      tC ← EstSpdupEff(sample_times, SE_min)
7      if tC < C_max then
8          break
9      else
10         l_p ← l_p − 1
11         C_max ← tC
12 l_p ← l_p + 1
```

Algorithm 3: ACME Automatic Tuning

6.3.1 Estimating Serial Execution Time

From the previous analysis, it follows that the workload frequency distribution of a good space decomposition should be similar to the one in Figure 9(a). It should contain a lot of fairly small tasks, in order to achieve load balance, but should avoid very small ones, since they tend to be false positives. Consequently, our optimization process is based on the following heuristic:

Observation 6 *A near-optimal partitioning will produce tasks with a workload frequency distribution that resembles a Gamma [23] distribution.*

A Gamma distribution Γ is characterized by a shape parameter α and a scale parameter β. Recall that line 4 of Algorithm 3 generates a sample of tasks for prefix length l_p. According to our heuristic, we assume that the sample approximates Γ. Therefore, we can use the sample to calculate approximations for the mean μ_Γ and standard deviation σ_Γ of Γ. Then, we calculate α and β as follows[4] [23]:

$$\alpha = \frac{\mu_\Gamma^2}{\sigma_\Gamma^2}, \qquad \beta = \frac{\mu_\Gamma}{\alpha} \tag{2}$$

The probability density function (PDF) of Γ is defined as:

$$\Gamma(x; \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{(\alpha-1)!} \tag{3}$$

As an example, consider the same settings as in Figure 9(a), decompose the space at $l_p = 3$ and draw a sample of 160 tasks. Figure 11 depicts the PDF of the runtime of the sample as solid bars, and the PDF of the estimated Γ distribution as dotted line. Observe that the PDF of Γ resembles closely the desired workload frequency distribution of Figure 9(a).

Let $\Lambda(t_i, t_j)$ be the expected number of tasks (in the entire space for a given l_p) with runtime between t_i and t_j. Given Γ, Λ is calculated as follows:

$$\Lambda(t_i, t_j) = |\Sigma|^{l_p} \int_{t_i}^{t_j} \Gamma(x; \alpha, \beta)\, dx \tag{4}$$

The serial execution time T_1 is the summation of the the execution times of all tasks. The lower bound of runtime for a task is zero, but the upper bound is

[4] The family of Gamma distributions contains many different shapes. To verify that our sample generates the desired leptokurtic, positively skewed non-symmetric distribution, we check: $4 > \alpha > 1$ and $\beta < \alpha$.

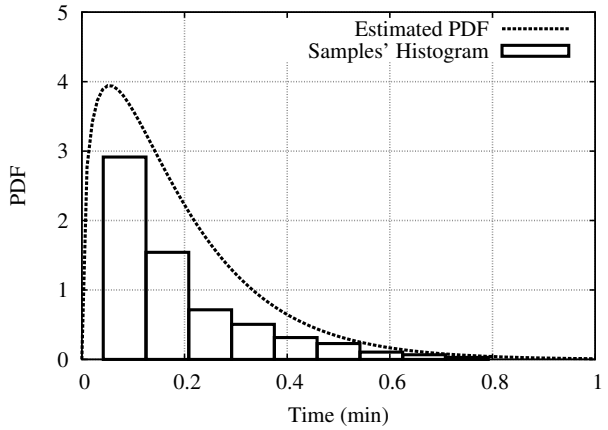


Fig. 11 Estimated probability density function (PDF) for the tasks workload frequency and the histogram from the actual execution times of 160 sample tasks.

unknown. Let x be an integer time unit. Then T_1 is defined as:

$$T_1 = \sum_{x=0}^{\infty} \frac{2x+1}{2} \Lambda(x, x+1) \tag{5}$$

6.3.2 Estimating Parallel Execution Time

We employ the queuing theory [15] to estimate the parallel execution time $T_{\mathcal{C}}$. We model the motif extraction process as a finite-source queue of $|\Sigma|^{l_p}$ tasks served by $\mathcal{C}$ servers (i.e., CPUs). Without loss of generality, we assume homogeneous servers. Since our population is finite, numerically simulating the queue provides an accurate representation of the real system [21]. ACME implements a discrete event simulator. We start with all tasks in the queue as tasks are generated by simply enumerating prefixes. The workloads of the tasks follow the workload frequency distribution of our sample prefixes. Equation 4 is used to create bins of tasks. The servers randomly consume tasks from different workload bins until all bins are empty. The output of the simulator is our estimation for the parallel execution time $T_{\mathcal{C}}$.

6.4 ACME Elasticity

The elasticity model of ACME furnishes users with an accurate estimation of the minimum amount of resources required to process a query within specific constraints. User constraints may involve the maximum allowed execution time; maximum amount of CPU hours, if the system is deployed in a typical shared research computing infrastructure; or a limit on the financial cost, if a commercial cloud computing provider is used.

Input: Sample times *sample_t*, *user_constraints*
Output: Suggested number of cores $\mathcal{C}_p$, estimated parallel time $T_{\mathcal{C}}$

```
// estimate PDF from sample execution times
1 α ← (MEAN(sample_t)/STDEV(sample_t))^2
2 β ← MEAN(sample_t)/α
// predict serial time left
3 T_1 ← Σ_{t=0}^{∞} ((2t+1)/2 Λ(t, t+1))
// predict parallel time and utilization
4 while user_constraints ≠ TRUE do
5     SETUPQUEUE(user_constraints, C_p)
6     (T_C, C_p) ← SIMULATEQUEUE(sample_t)
```

Algorithm 4: ACME Elasticity Model

Algorithm 4 describes the elasticity model. It takes the execution time of each of the tasks in the random sample (see Section 6.3) and the user constraints as input, and outputs the number of CPUs to use, together with the estimated time and speedup efficiency. In line 5, a queue is setup by randomly taking tasks with workloads according to our probability density function. The execution of the query is simulated using a certain number of CPUs. This simulation is done in a loop where the number of CPUs is varied until the user constraints are met.

Given the expected performance variability on public clouds [27], users should be able to reevaluate the situation online and adapt accordingly. Our serial time estimation may be reevaluated at runtime to guide user decisions and meet their constraints. A slight modification of Equation 4 is used to account only for the tasks not executed yet. We substitute $|\Sigma|^{l_p}$ with $|\Sigma|^{l_p} - k$, where k is the number of already completed tasks.

The output of our model can be used in many ways. For example, if the pricing scheme of a cloud computing provider is given, our model can predict accurately the expected financial cost. We present such a case study in Section 8.1.2.

7 Cache-optimized Motif Extraction (CAST)

ACME decomposes the search space into sub-tries of arbitrary sizes. Each sub-trie is maintained independently using our cache-optimized mechanism, called CAST[5].

7.1 Spatial and Temporal Memory Locality

Existing motif extraction methods realize the search space trie as a set of nodes, where each node has a one

[5] CAST stands for Cache Aware Search space Traversal

character label, pointers to its parent and children, and its occurrences set. These nodes are dynamically allocated and deallocated. The maximum number of 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. These nodes are scattered in main memory and visited back and forth to traverse all motif candidates. Consequently, existing methods suffer dramatically from cache misses, plus the overhead of memory allocation and deallocation.

A branch of trie nodes represents a motif candidate as a sequence of symbols. These symbols are conceptually adjacent with preserved order; allowing for spatial locality. Moreover, maintaining occurrences sets is a pipelined process; for instance, the occurrences set of `AA` is used to build the occurrences set of `AAA`. This leads to temporal locality. Existing approaches overlooked these important properties.

We propose CAST, a representation of the search space trie together with access methods, that is specifically designed to be cache efficient by exhibiting spatial and temporal locality. 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 experimented with DNA, protein, and English sequences of gigabyte sizes, where the longest frequent motif lengths are 28, 95 and 42 symbols respectively. Moreover, the occurrences set is also realized as an array. A cache of a modern CPU can easily fit a sub-trie branch and, in most cases, its occurrences array. For temporal locality, once we construct the occurrences array $\mathcal{L}(v_i)$ of branch node v_i, we revisit each occurrence to generate $\mathcal{L}(v_{i+1})$. We take advantage of the fact that the total frequency of $\mathcal{L}(v_{i+1})$ is bounded by that of $\mathcal{L}(v_i)$. Therefore, with high probability all data necessary for the traversal and validation are already in the cache.

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 occur in branch[i − 1].L do
10         if occur is a full suffix tree path label then
               // check child nodes in suffix tree
11             foreach child of occur.T do
12                 if first symbol in child label ≠ next
                   then
13                     child.D = occur.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 occur.T label ≠ next
               then
22                 increment occur.D
23                 if occur.D > d then
24                     Discard(occur)
25                     if branch[i].F < f then
26                         Prune(branch[i])
27             else
28                 add occur to branch[i].L
29     if isValid(branch[i]) then Output(branch[i])
30     i++
31     next ← DepthFirstTraverse(i)
```

Algorithm 5: CAST Motifs Extraction

7.2 CAST Algorithm

CAST extracts valid motifs as follows: (*i*) initialize the sub-trie prefix; then (*ii*) extend the prefix as long as it leads to valid motif candidates; otherwise (*iii*) prune the extension. In the rest of this section we consider sequence S from Figure 3 and use an example query Q with $\sigma = 12$, $l_{min} = l_{max} = 5$ and $d = 2$.

Algorithm 5 shows the details. Let *branch* be the sub-trie branch array. An element $branch[i]$ contains a symbol c, an integer F, and a pointer, as shown in Figure 12. Each sub-trie has a prefix p that is extended to recover all motif candidates sharing p. $branch[i]$ represents motif candidate $m_i = pc_1 \ldots c_i$, where c_i is a symbol from level i of 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 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 $\mathcal{L}(p)$.

7.2.1 Prefix Initialization

Algorithm 5 starts by creating the occurrences array of the given fixed-length prefix before recovering motif candidates. CAST commences the occurrences array

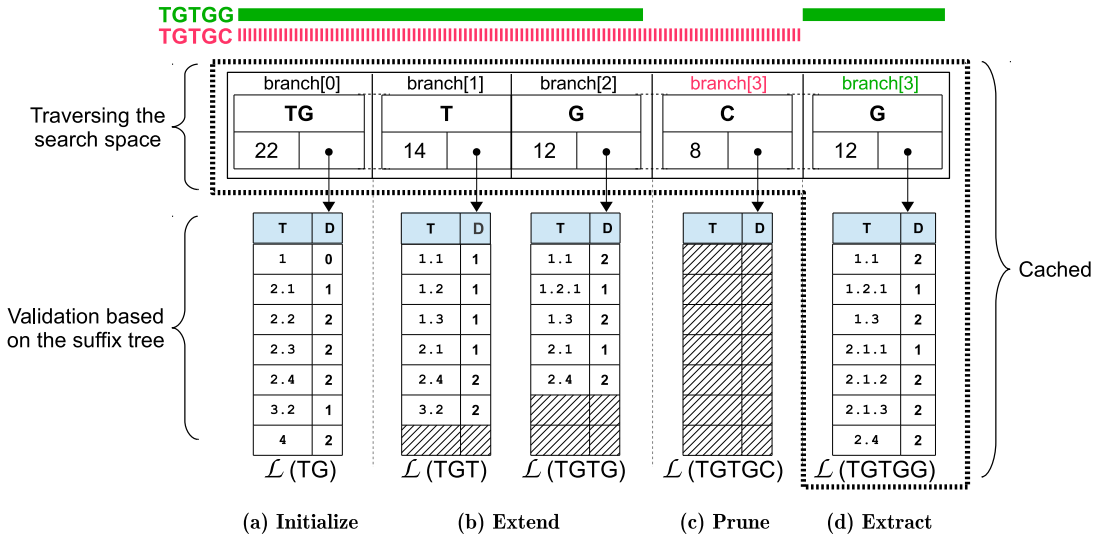


Fig. 12 Snapshot of CAST processing for $Q'(|S| = 23, \sigma = 12, l_{min} = 5, l_{max} = 5, d{=}2)$ over the sequence of Figure 3. Prefix `TG` is extended one symbol at a time to maintain `TGTGC` and `TGTGG` branches. A branch is traversed from ancestor to descendant by moving from left to right. CAST array (*branch*) and the occurrences array of the deepest descendant are easily cached, since both fit into small contiguous memory blocks.

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

For example, the sub-trie with prefix `TG` is initialized by CAST in two phases using the suffix tree in Figure 3. Figure 12(a) shows the final set $\mathcal{L}$(`TG`) of occurrences 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 sum of frequencies of the suffix tree nodes in $\mathcal{L}$(`TG`): $7 + 5 + 5 + 2 + 1 + 1 + 1 = 22$.

7.2.2 Extension, Validation and Pruning

Since `TG` is frequent enough, it is extended by traversing its search space sub-trie. The depth-first traversal (DFT) of the sub-trie starts at line 5 in Algorithm 5 to extend *branch*[0]; it considers all symbols of Σ at each level of the DFT. At level i, DepthFirstTraverse returns c_i to extend *branch*$[i-1]$. Figure 12(b) demonstrates the extension of *branch*[0] with symbols `T` then `G`.

The maintenance of the occurrences set is a pipelined function, where $\mathcal{L}(branch[i+1])$ is constructed from its parent's, $\mathcal{L}(branch[i])$. This process is done in the loop starting at line 9. 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 in Figure 12(b), Figure 12(c), and Figure 12(d), respectively.

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 12(c), the total frequency of `TGTGC` drops below the frequency threshold $\sigma = 10$ after discarding node `1.1` of frequency 4 from $\mathcal{L}$(`TGTG`), i.e. $12 - 4 < \sigma$. Since `TGTGC` has frequency less than σ, we do not need to check the rest of the occurrences and the branch is pruned (see line 16 in Algorithm 5).

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

Table 3 The specifications of the various systems used in the experiments discussed in the evaluation of ACME.

#	Architecture	Cores	RAM	L1 cache	L2 cache	L3 cache
1	32-bit Linux machine	2 cores @ 2.16GHz	2GB shared	64KB	1MB	
2	64-bit Linux machine	12 cores @ 2.67GHz	192GB shared	64KB	256KB	12MB
3	64-bit Linux SMP	32 cores @ 2.27GHz	624GB shared	64KB	256KB	24MB
4	Amazon EC2 64-bit Linux cluster	40 on-demand large instances, each having 2 cores	7.5GB each 300GB total	64KB	6MB	
5	IBM Blue Gene/P supercomputer	16,384 quad-core PowerPC processors @ 850MHz	4GB each 64TB total	64KB	2KB	8MB
6	64-bit HPC Linux cluster	480 cores @ 2.1GHz	6GB each 3TB total	128KB	512KB	5MB

CAST supports exact-length motifs, maximal motifs, and supermaximal 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 supermaximal motifs, IsValid passes the right-supermaximal motifs to a combiner that implements Algorithm 1 as discussed in Section 4.

8 Evaluation

We implemented ACME [6] in C++. We adopted two different models: (*i*) ACME-MPI uses the message passing interface (MPI) to run on shared-nothing systems, such as clusters and supercomputers; and (*ii*) ACME-THR utilizes threading on multi-core systems, where the sequence and its suffix tree are shared among all threads.

We conducted comprehensive experiments to analyze the cloud-oriented features of ACME, and to compare with existing methods on different motif types, scalability and computational efficiency. The query workload is not only affected by sequence size but also by alphabet size, motif length, distance and frequency. In our experiments, we used: (*i*) data-intensive queries, where the sequence size is in the order of few gigabytes, (*ii*) processing-intensive queries, where the thresholds are loose and lead to huge search space, and (*iii*) a combination of both cases.

We use real datasets of different alphabets: (*i*) DNA[7] of the entire human genome (2.6GB, 4 symbols alphabet); (*ii*) Protein[8] sequence (6GB, 20 symbols); and (*iii*) English[9] text from an archive of Wikipedia (1GB, 26 symbols). In some experiments, especially in cases where our competitors are too slow, we use only a prefix of these datasets. We deployed ACME on various systems with different architectures; the details appear in Table 3. In each experiment, we refer to the used system by its serial number, e.g., *System#5* for the supercomputer.

8.1 Analyzing ACME Cloud-Oriented Features

In this section, we evaluate the cloud-oriented features of ACME, namely automatic tuning, elasticity, parallel scalability, and cache efficiency.

8.1.1 Automatic tuning: accuracy and cost

First we evaluate the accuracy and overhead of our automatic tuning process. We aim at finding the best sample size that achieves high accuracy with an acceptable overhead. The experiments in this section were run on *System#5* (refer to Table 3).

First, we search exhaustively for the best prefix length l_p for our query on a DNA sequence. We run the entire query for varying prefix length and for varying number of cores. The speedup efficiency SE for all combinations is shown in Table 4. The best SE is achieved for $l_p = 9$. Shorter prefixes generate too few tasks that cannot achieve load balance, especially when scaling to thousands of cores; whereas longer prefixes result in too many false positives.

Next, we use ACME's automatic tuning on the same sequence and query. We vary the sample size used in Algorithm 3 to study its effect on accuracy. Table 5 shows the results. The suggested value for l_p converges quickly. Our algorithm needs roughly 160 sample tasks to stabilize to the optimal value $l_p = 9$. Note that the overhead of running this sample is less than 10sec, compared to

[6] ACME code and the used datasets are available online at: http://cloud.kaust.edu.sa/Pages/acme_software.aspx

[7] http://webhome.cs.uvic.ca/ thomo/HG18.fasta.tar.gz

[8] http://www.uniprot.org/uniprot/?query=&format=*

[9] http://en.wikipedia.org/wiki/Wikipedia:Database_download

Table 4 Speedup efficiency with different prefix lengths l_p on *System#5*. Query over DNA sequence with serial execution of 5.2 hours. For each combination the complete Q was executed.

$Q(|S|$=8MB, σ=10K, l_{min}=12, l_{max}=∞, d=3)

Cores	Speedup Efficiency SE			
	l_p=7	l_p=8	l_p=9	l_p=10
512	0.94	0.97	**0.98**	0.81
1,024	0.87	0.97	**0.97**	0.83
2,048	0.83	0.92	**0.97**	0.83
4,096	0.46	0.76	**0.92**	0.76
8,192	0.25	0.46	**0.76**	0.46

Table 5 Sensitivity analysis of the automatic tuning sample size. The best prefix length for this query is 9, as computed exaustively in Table 4. Having between 100 and 200 samples finds the accurate prefix length l_p. The overhead is only a few seconds as opposed to 5.4 hours to generate Table 4.

$Q(|S|$=8MB, σ=10K, l_{min}=12, l_{max}=∞, d=3)

Sample size	Tuning overhead (sec)	Suggested l_p
10	9.0	7
20	11.0	7
40	6.1	8
80	3.7	**9**
160	6.1	**9**
320	11.4	**9**

5.4 hours for generating exhaustively the values for the previous experiment (i.e., Table 4). Observe that: (*i*) the overhead does not depend directly on the sample size, but on the actual workload of tasks. A small sample can contain prefixes of very high workload compared to those in a larger sample, and vice versa. Moreover, the number of iterations until the tuning algorithm converges depends on the selected tasks; not on the sample size. (*ii*) We run the tuning process on one core; however, the entire process can be efficiently parallelized.

8.1.2 Elasticity model: deployment on Amazon EC2

ACME's elasticity model estimates the serial and parallel execution times, and the speedup efficiency in order to decide the minimum amount of resources that meet the user's constraints. Below we discuss the accuracy and cost of the estimation, and demonstrate the applicability of our model on Amazon EC2. Our experiments were run on *System#4* (refer to Table 3).

Table 6 shows the estimated serial execution time, the error compared to the actual serial execution time, and the estimation overhead. Our estimation is very accurate (i.e., less than 1.5% error), using very small samples, in the order of 160 to 320 tasks. The overhead of the estimation process is also very small; it did not exceed 10sec in the worst case. Based on Table 6, we set the default sample size to 200 tasks for the rest of the paper. We also evaluated the accuracy of the parallel execution time estimation, by varying the number of cores. Figure 13 shows that our estimation is very close to the actual parallel run time.

Table 6 Sensitivity analysis of the sample size in estimating serial execution time.

$Q(|S|$=16MB, σ=60K, l_{min}=8, l_{max}=∞, d=3)

Sample Size	Overhead (sec)	Estimated Time (sec)	Error (%)
10	5.1	2,792	10.58
20	4.1	2,028	19.67
40	4.0	2,867	13.57
80	7.8	2,233	11.53
160	6.7	2,494	**1.22**
320	10.0	2,487	**1.49**

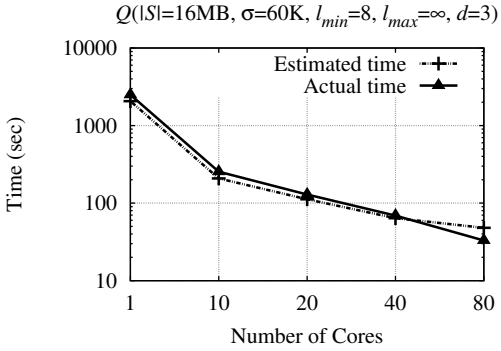


Fig. 13 Accuracy of the estimation of the parrallel execution time, used for automatic tuning and elasticity.

The next experiment demonstrates a scenario in which an Amazon EC2 user needs to know how many instances to rent in order to query the DNA dataset. Assume the user needs the query to be executed in less than 3 hours without spending more than $20. Each Amazon EC2 instance costs $0.24 per hour[10]. We employ our elasticity model to estimate the expected runtime and financial cost for varying number of EC2 instances. The results are summarized in Table 7. For example, if 10 instances are rented the financial cost constraint is satisfied (i.e., $14.40 < 20$), but the expected runtime is 5.9 hours, much longer than the user's request. The runtime constraint is satisfied for 30 instances; however, the financial cost constraint is not. Interestingly, if *more* instances are rented (i.e., 40), then both constraints are satisfied. This happens because of the coarse-grained pricing schemes imposed by most cloud service providers; Amazon, for instance, charges on hourly basis. Because of the pricing scheme, the runtime estimation needs to be accurate, else the user may face higher costs. In our example, if the user rents 50 instances, our model estimates around one hour execution time, costing $12. However, if the actual execution takes a few seconds more than an hour, the cost will

[10] http://aws.amazon.com/ec2/pricing/

Table 7 Deployment on Amazon EC2, using the DNA dataset. Because of user constraints, runtime and financial cost must not exceed 3 hours and $20, respectively. Both constraints are satisfied if 40 instances are rented.

$Q(|S|$=32MB, σ=60K, l_{min}=9, l_{max}=∞, d=4)

	Number of Amazon EC2 Instances				
	1	**10**	**20**	**30**	**40**
Cores	2	20	40	60	80
Cost/Hour	$0.24	$2.40	$4.80	$7.20	$9.60
Est. Time	2.5 Days	5.9 Hr	3.1 Hr	2.1 Hr	**1.5 Hr**
Est. Cost	$14.40	$14.40	$19.20	$21.60	**$19.20**
Act. Time	2.9 Days	5.1 Hr	4.3 Hr	2.3 Hr	1.6 Hr
Act. Cost	$16.80	$14.40	$24.00	$21.60	$19.20

double to $24. Because of shared cloud resources, such performance variability is expected. Nevertheless, our experiments demonstrate that the estimations from our model are very close to the actual values in most cases; the results are shown at the lower part of Table 7.

8.1.3 Startup cost and elasticity overhead

In this section we show the startup cost of ACME and the overhead of elastically changing the number of workers during execution. We conducted the experiments on *System#6* (refer to Table 3), using the DNA dataset. We run the same query using 5, 50 and 350 cores. The results are shown in Table 8. We break the total execution time into: (*i*) Startup time. It is the sum of time to transfer the input sequence S to all new workers and construct the corresponding suffix tree, independently in each worker. Startup time increases only slightly when more cores are used: when the number of cores increases by 70x, startup time increases only by 1.3x. Since each worker builds independently and in parallel its local suffix tree, the increase is due to network congestion (i.e., S is sent by the master to all workers). Note that the most significant portion of the startup overhead is suffix tree construction. Its complexity is linear to S, and does not depend on the query. (*ii*) Extraction time. It is the time to run the actual motif extraction process and drops dramatically with more cores, as expected. Observe that, in all cases, the startup overhead is orders of magnitude less than the extraction time.

ACME can scale elastically in or out by removing or adding workers, respectively. Scaling in does not incur any overhead, because tasks are scheduled to workers independently in a pull fashion. During scaling out, on the other hand, new workers incur the startup overhead. As discussed, the startup overhead is not significant (i.e., orders of magnitude smaller than the useful work). Moreover, the startup overhead occurs independently on each new worker; existing workers continue processing without interference.

Table 8 Startup time is the time to transfer S to each worker, plus the time to construct the suffix tree. It increases only slighlty with the number of workers, and is orders of magnitude less than the extraction time.

$Q(|S|$=32MB, σ=60K, l_{min}=9, l_{max}=∞, d=4)

Cores	**Startup time**	**Extraction time**
5	49 sec	2 days
50	50 sec	4 hours
350	63 sec	50 minutes

Table 9 Scalability of PSmile on *System#6* using the DNA dataset. The speedup efficiency of PSmile is hindered by load imbalance due to improper search space partitioning and static scheduling ($SE < 0.8$ is considered low).

$Q(|S|$=32MB, σ=10K, l_{min}=10, l_{max}=15, d=3)

	Time (sec)		**Speedup Efficiency**	
Cores	**PSmile**	**ACME**	**PSmile**	**ACME**
5	19,972	18,883	1.00	1.00
10	9,894	8,476	0.90	0.99
20	4,869	3,978	0.86	0.99
40	2,786	1,969	0.74	0.98
80	1,787	989	0.57	0.97
160	1,130	580	0.44	0.82

8.1.4 Parallel scalability

This section investigates ACME's parallel scalability. We test the so-called strong-scalability, where the number of cores is increased while the problem size is fixed. We first compare ACME against PSmile, the only parallel competitor. PSmile uses grid-specific libraries to parallelize a previous sequential motif extraction algorithm. Calculating the speedup efficiency from the experiments reported in the PSmile paper, speedup efficiency SE drops to 0.72 when using 4 nodes only; recall that in practice $SE < 0.8$ is considered low. We suspected that the bad performance was partially due to inefficient implementation. For fairness we implemented the search space partitioning and task scheduling scheme of PSmile within ACME, utilizing our cache-efficient trie traversal algorithms.

Table 9 shows the results, using our optimized implementation of PSmile. The experiment was run on *System#6* (refer to Table 3). Due to resource management restrictions, the minimum number of cores used in this experiment was 5; therefore, speedup efficiency is calculated relative to a 5-core system. PSmile does not scale efficiently not even on 40 cores, due to problematic space partitioning and scheduling, which creates load imbalance. In contrast, for this particular query, ACME scales easily to more than 160 cores.

The next experiment investigates ACME's scalability to the extreme, by utilizing up to 16,384 cores on a

Table 10 Scalability of ACME on a supercomputer for the Protein dataset. ACME scales to tens of thousands of cores with high speedup efficiency.

$Q(|S|$=32MB, σ=30K, l_{min}=12, l_{max}=∞, d=3)

Cores	Time (Hrs.)	Speedup Efficiency
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

Table 11 Supermaximal Motifs from the complete DNA for the human genome (2.6GB) categorized by length. The total number of supermaximal motifs is more than total number of exact-length motifs.

$Q(|S|$=2.6GB, σ=500K, l_{min}=15, l_{max}=var, d=3)

Supermaximal ($l_{max} = \infty$)						Exact-length ($l_{max} = l_{min}$)	
Len	Count	Len	Count	Len	Count	Len	Count
15	359,293	20	30,939	25	443	15	446,344
16	82,813	21	33,702	26	143		
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			Total	446,344

Table 12 Analysis of three sequences of different alphabets, each of size 1GB.

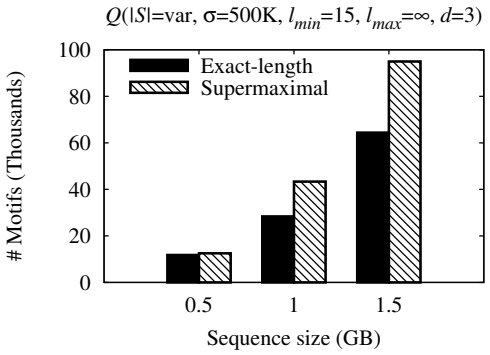
	Query	Motifs	Longest	Time
DNA	σ=500K, l=12−∞, d=2	5,937	20	0.6 m
Protein	σ=30K, l=12−∞, d=1	96,806	95	2.1 m
English	σ=10K, l=12−∞, d=1	315,732	42	3.5 m

supercomputer. We use the Protein dataset, which results to a much larger search space than DNA because of the larger alphabet (i.e., 20 symbols). With larger alphabets, ACME automatic tuning model will suggest a small prefix length; in this case $l_p = 5$. The experiment was run on *System#5*. Due to resource management restrictions, the minimum number of cores used in this experiment was 256 cores; hence, speedup efficiency is calculated relatively to a 256-core system. The results are shown in Table 10 and demonstrate the excellent scalability of ACME to thousands of cores. On 256 cores, the query takes almost 20 hours to execute; whereas with 16,384 cores it finishes in only 18.6min, achieving almost perfect (i.e., 0.98) speedup efficiency. It is worth mentioning that the same query on a high-end 12-core workstation (i.e., *System#2*) takes more than 7 days. Recall that each core of the workstation is much faster (i.e., 2.67GHz) than a supercomputer core (i.e., 850MHz).

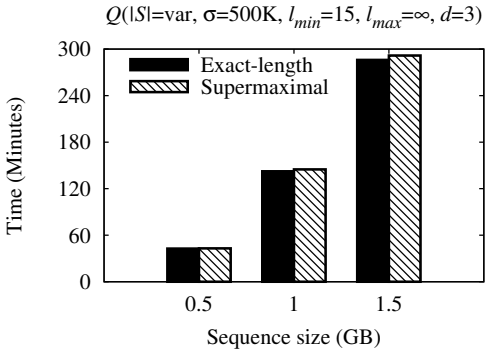
Table 13 The overhead of extracting supermaximal motifs over maximal motifs is not critical due to ACME's pipelined strategy for filtering motifs that are subsequences of others.

$Q(|S|$=1.5GB, σ=500K, l_{min}=15, l_{max}=∞ d=3)

	Time	Motifs
Maximal motifs	303.7 min	144,952
Supermaximal	313.7 min	87,680
Difference	10 min	57,272
Percentage	3.3%	39.5%



(a) Number of motifs



(b) Time performance

Fig. 14 Supermaximal vs Exact-length motifs extraction using ACME.

8.2 ACME Comprehensive Motif Extraction Support

In addition to supermaximal motifs, ACME extracts maximal and exact-length ones. The following paragraphs evaluate ACME's scalability in terms of input size and query complexity; and compare ACME against state of the art systems for maximal and exact-length motifs.

8.2.1 Gigabyte-long sequences and varying alphabets

The experiments discussed in this section were run on *System#2* and *System#3* (refer to Table 3). Table 11 shows the count of all supermaximal motifs (i.e., no bound for l_{max}), grouped by length, that appear at

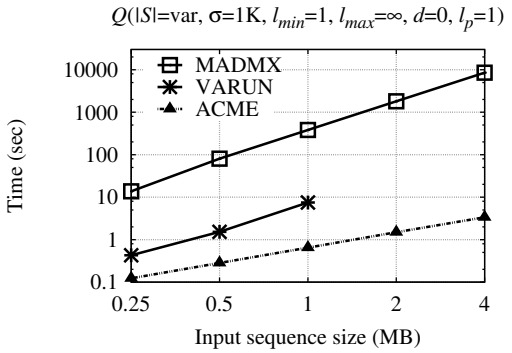


Fig. 15 Serial execution of ACME extracting maximal motifs using one core vs MADMX and VARUN.

least $\sigma = 500K$ times in the entire human DNA (i.e., 2.6GB). For reference, the count of all *maximal* motifs with length 15 is also shown. The longest supermaximal motif is 28 symbols long. This means that the CAST array size did not exceed 252 bytes in a 32-bit system (28 elements of 9 bytes each). With current CPU cache sizes, not only the CAST array will fit in the cache but most probably the occurrences array too. Consequently, ACME handles the extra workload of extracting maximal and supermaximal motifs efficiently.

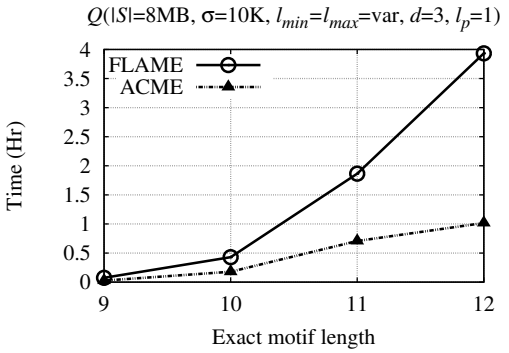
Observe that, in the entire human genome, there are around 20% more supermaximal motifs of length 15 and more, compared to the number of motifs with exact length 15. Figure 14(a) shows the corresponding counts by varying the size of the input sequence (i.e., using prefixes of the entire DNA). The number of supermaximal motifs is in all cases significantly more than the exact-length ones. Figure 14(b) compares the time to extract exact-length versus all supermaximal motifs. The difference is negligible (i.e., around 4%), confirming the efficiency of our supermaximal extraction algorithm.

ACME supports different alphabet sizes. Table 12 shows the results of extracting supermaximal motifs from 1GB sequences of different alphabets. We also extracted maximal and supermaximal motifs from 1.5GB of the DNA sequence. Table 13 shows that ACME's pipelined strategy for filtering motifs that are subsequences of others introduces an overhead of 3.3% over the maximal motifs extraction time. In this process, about 40% of the maximal motifs are discarded because they are subsequences of other ones.

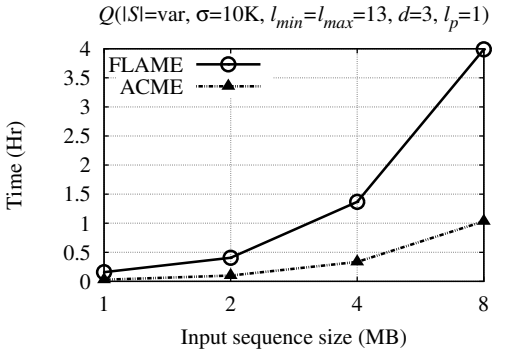
8.2.2 Comparison against state-of-the-art

We compared ACME against FLAME, MADMX, and VARUN. Since the source code for FLAME was not available, we implemented it using C++. MADMX[11]

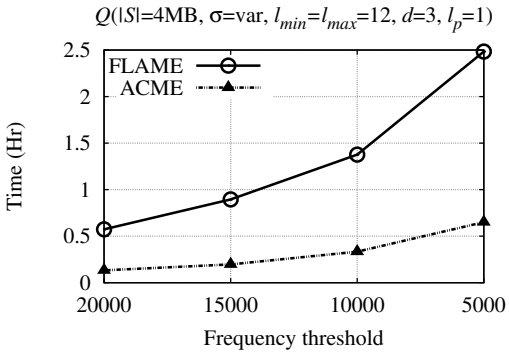
[11] http://www.dei.unipd.it/wdyn/?IDsezione=6376



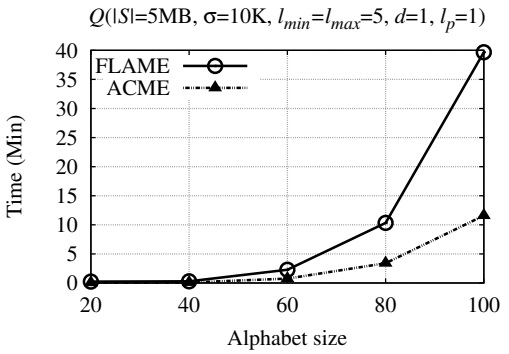
(a) Variable motif length



(b) Variable sequence size



(c) Variable frequency



(d) Variable alphabet size

Fig. 16 Serial execution of ACME extracting exact length motifs using one core vs FLAME. ACME is superior as the workload is increased using different factors.

and VARUN[12] are available from their authors' web sites. These systems do not support parallel execution and are restricted to particular motif types. The following experiments were executed on *System#2*. Since our competitors run serially, for fairness ACME uses only one core. The reported time includes the suffix tree construction and motif extraction time; the former is negligible compared to the extraction time. Note that we use small datasets (i.e., up to 8MB from DNA), because our competitors cannot handle larger inputs.

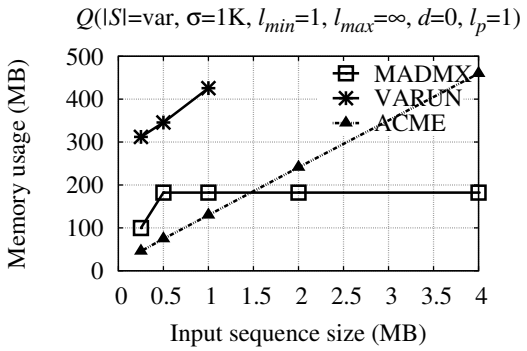
ACME is evaluated against MADMX and VARUN when extracting maximal motifs. Different similarity measures are utilized by ACME, MADMX and VARUN. Therefore, this experiment does not allow mismatches (i.e., $d = 0$) in order to produce the same results. Since the workload increases proportionally to the distance threshold, this experiment is relatively of light workload. Figure 15 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). We were not able to test the scalability of VARUN and MADMX in terms of alphabet size because they support DNA sequences only.

FLAME and ACME produce identical exact-length motifs. The serial execution of ACME significantly outperforms FLAME with increasing workload, as illustrated in Figure 16. We vary the workload by increasing motif length (Figure 16(a)), sequence size (Figure 16(b)), frequency threshold (Figure 16(c)), and alphabet size (Figure 16(d)). The impressive performance of our system is a result of ACME's cache efficiency. Note that, if we were to allow ACME to utilize all cores, then it would be one more order of magnitude faster. For example, we tested the query of Figure 16(a) when motif length is 12: FLAME needs 4 hours, whereas parallel ACME finishes in 7 minutes. The alphabet size experiment was run using synthetic datasets, generated with random distribution to guarantee comparative workloads between sequences of different alphabets.

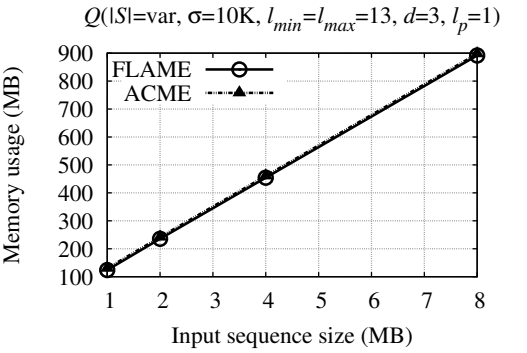
8.2.3 Memory usage

The memory usage of ACME grows linearly with respect to the sequence size. This is because the main factor is the suffix tree index; its implementation is not optimized since it is not a contribution of ACME. Figure 17(a) shows that the memory footprint of VARUN is three times higher than ACME's. MADMX has a low and constant memory footprint for sequences over 0.5MB but at the expense of 2 orders of magnitude higher runtime (see Figure 15). Figure 17(b) shows that FLAME and ACME have the same memory footprint because they share the same suffix tree implementation. Yet, ACME performs better than FLAME because of our cache-efficient approach, CAST (see Figure 16(b)).

[12] http://researcher.ibm.com/files/us-parida/varun.zip



(a) ACME vs MADMX and VARUN



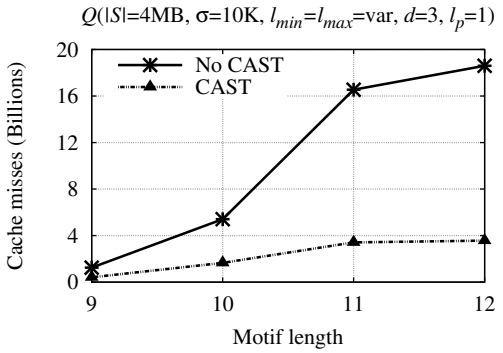
(b) ACME vs FLAME

Fig. 17 Memory usage of ACME compared to VARUN and MADMX, for the DNA dataset and the queries from Figure 15 and Figure 16(b).

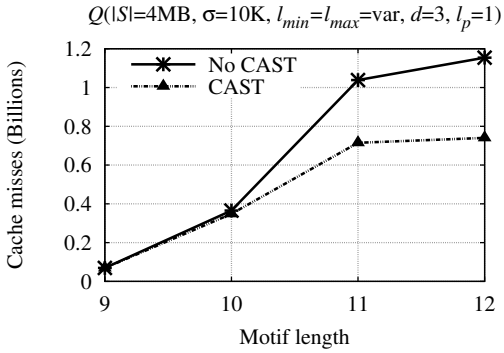
8.3 Cache efficiency

Existing motif extraction methods incur a lot of cache misses while traversing the search space. ACME 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 mechanism, as *NoCAST*.

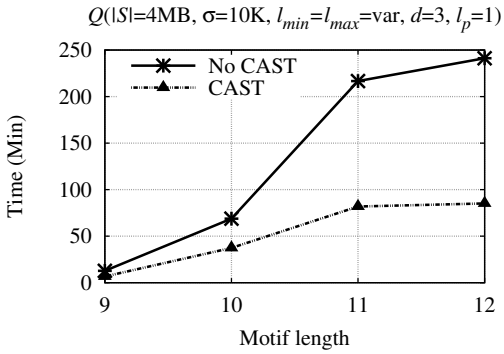
We used the *perf* Linux profiling tool to measure the L1 and L2 cache misses. This test was done on *System#1* (refer to Table 3). CAST significantly outperforms NoCAST in terms of cache misses and execution time especially when the motif length, and consequently the workload, is increased, as shown in Figure 18. The



(a) L1 cache misses



(b) L2 cache misses



(c) Time performance

Fig. 18 Correlation between caches misses and motif extraction time; DNA dataset.

difference between CAST and NoCAST shows earlier in L1 cache. However, the difference in L2 cache misses starts to show as the motif length is increased and has the same pattern. The correlation between cache efficiency (Figure 18(a) and Figure 18(b)) and serial execution time (Figure 18(c)) is clear.

9 Conclusion

In this paper we introduced ACME, a cloud-oriented combinatorial method for extracting supermaximal motifs from a single long sequence, which is required in a variety of applications, including bioinformatics, web log analysis, time series and others. In addition, ACME supports exact-length and maximal motifs. ACME arranges the search space in contiguous blocks that take advantage of the cache hierarchy in modern architectures. Moreover ACME supports large scale parallelism and introduces an automatic tuning mechanism that estimates the expected execution time for various scenarios and decides a good decomposition of the search space that leads to near-optimal resource utilization. Automatic tuning is particularly useful for cloud environments, since it suggests the minimum amount of resources required (i.e., minimizes financial cost), while meeting a user-defined execution time constraint. In our experiments we demonstrated that ACME handles the entire DNA sequence for the human genome on a single high-end multi-core machine; this is 3 orders of magnitude longer compared to the state-of-the-art. We also showed that ACME can be deployed in a variety of large scale parallel architectures, including Amazon EC2 and a supercomputer with 16,384 CPUs.

This work is part of a research project for developing a generic framework that supports automatic tuning and elasticity for parallel combinatorial search algorithms. Our future work also includes the development of a disk-based version of ACME to support longer sequences in systems with limited memory.

References

1. Apostolico, A., Comin, M., Parida, L.: VARUN: discovering extensible motifs under saturation constraints. IEEE/ACM Transactions on Computational Biology Bioinformatics **7**(4), 752–26 (2010)
2. Becher, V., Deymonnaz, A., Heiber, P.: 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. Carvalho, A.M., Oliveira, A.L., Freitas, A.T., Sagot, M.F.: A parallel algorithm for the extraction of structured motifs. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 147–153 (2004)
4. Challa, S., Thulasiraman, P.: Protein sequence motif discovery on distributed supercomputer. In: Proceedings of the international conference on Advances in grid and pervasive computing (GPC), pp. 232–243 (2008)
5. Das, M.K., Dai, H.K.: A survey of DNA motif finding algorithms. BMC Bioinformatics **8**(S-7), S21 (2007)
6. Dasari, N.S., Desh, R., Zubair, M.: An efficient multicore implementation of planted motif problem. In: Proceedings of the International Conference on High Performance Computing and Simulation (HPCS), pp. 9–15 (2010)
7. Dasari, N.S., Ranjan, D., Zubair, M.: High performance implementation of planted motif problem using suffix trees. In: Proceedings of the International Conference on High Performance Computing and Simulation (HPCS), pp. 200–206 (2011)

8. Federico, M., Pisanti, N.: Suffix tree characterization of maximal motifs in biological sequences. Theoretical Computer Science **410**(43), 4391–4401 (2009)
9. Floratou, A., Tata, S., Patel, J.M.: Efficient and Accurate Discovery of Patterns in Sequence Data Sets. IEEE Transactions on Knowledge and Data Engineering (TKDE) **23**(8), 1154–1168 (2011)
10. Grossi, R., Pietracaprina, A., Pisanti, N., Pucci, G., Upfal, E., Vandin, F., Salzberg, S., Warnow, T.: MADMX: A Novel Strategy for Maximal Dense Motif Extraction. In: Proceedings of Workshop on Algorithms in Bioinformatics, pp. 362–374 (2009)
11. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press (1997)
12. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1–12 (2000)
13. Huang, Yang, Chowdhary, Kassim, Bajic: An algorithm for ab initio dna motif detection. In: Information Processing and Living Systems, vol. 2, pp. 611–614 (2005)
14. Huang, C.W., Lee, W.S., Hsieh, S.Y.: An improved heuristic algorithm for finding motif signals in DNA sequences. IEEE/ACM Transactions on Computational Biology Bioinformatics **8**(4), 959–975 (2011)
15. Kleinrock, L.: Queueing Systems, vol. I: Theory. Wiley-Interscience (1975)
16. Liu, Y., Schmidt, B., Maskell, D.L.: An ultrafast scalable many-core motif discovery algorithm for multiple gpus. In: Proceedings of the International Symposium on Parallel and Distributed Processing, pp. 428–434 (2011)
17. Mabroukeh, N.R., Ezeife, C.I.: A taxonomy of sequential pattern mining algorithms. ACM Computing Surveys **43**(1), 1–41 (2010)
18. Mansour, E., Allam, A., Skiadopoulos, S., Kalnis, P.: Era: efficient serial and parallel suffix tree construction for very long strings. Proceedings of the VLDB Endowment **5**(1), 49–60 (2011)
19. Marchand, B., Bajic, V.B., Kaushik, D.K.: Highly scalable ab initio genomic motif identification. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 56:1–56:10 (2011)
20. Marsan, L., Sagot, M.F.: Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. Journal of Computational Biology **7**(3-4), 345–362 (2000)
21. Meisner, D., Wenisch, T.F.: Stochastic queuing simulation for data center workloads. In: Exascale Evaluation and Research Techniques Workshop (2010)
22. Mueen, A., Keogh, E.: Online discovery and maintenance of time series motifs. In: Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp. 1089–1098 (2010)
23. Papoulis, A., Pillai, S.U.: Probability, random variables, and stochastic processes. Tata McGraw-Hill Education (2002)
24. Sagot, M.F.: Spelling Approximate Repeated or Common Motifs Using a Suffix Tree. In: Proceedings of 3rd Latin American Symposium on Theoretical Informatics, pp. 374–390 (1998)
25. Sahli, M., Mansour, E., Kalnis, P.: Parallel motif extraction from very long sequences. In: Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM) (2013)
26. Saxena, K., Shukla, R.: Significant Interval and Frequent Pattern Discovery in Web Log Data. International Journal of Computer Science Issues **7**(1(3)), 29–36 (2010)
27. Schad, J., Dittrich, J., Quiané-Ruiz, J.A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proceedings of the VLDB Endowment **3**(1-2), 460–471 (2010)
28. Tsirogiannis, D., Koudas, N.: Suffix tree construction algorithms on modern hardware. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 263–274 (2010)
29. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
30. Xie, X., Mikkelsen, T.S., Gnirke, A., Lindblad-Toh, K., Kellis, M., Lander, E.S.: Systematic discovery of regulatory motifs in conserved regions of the human genome, including thousands of ctcf insulator sites. Proceedings of the National Academy of Sciences **104**(17), 7145–7150 (2007)
31. Yun, U., Ryu, K.H.: Approximate weighted frequent pattern mining with/without noisy environments. Knowledge-Based Systems **24**(1), 73–82 (2011)

A Proof for Algorithm 1

The set of supermaximal motifs from Algorithm 1 is correct and complete. Let SpellTrieFromLeaves be a function that takes as input a trie and returns the paths from the root to each leaf.

Lemma 1 SpellTrieFromLeaves *returns all sequences in the input trie that are not proper prefixes of any other.*

Proof By construction of the trie. □

We produce M_s in two steps: (i) we produce a right-supermaximal set of sequences M_{rs} by calling SpellTrieFromLeaves on the pruned search space, and (ii) we pass M_{rs} to Algorithm 1, whose job is to eliminate proper suffixes from M_{rs} to produce M_s. The outline of the proof is as follows: first we prove that SpellTrieFromLeaves eliminates proper prefixes from an input set of sequences, then we show that SpellTrieFromLeaves can be used to eliminate proper suffixes, after that we show that Algorithm 1, when given a right-supermaximal set of motifs M_{rs}, produces the supermaximal motifs set. We conclude our proof by showing that the input we give to Algorithm 1 is indeed right-supermaximal.

SpellTrieFromLeaves can be used to remove proper suffixes from a set of sequences. When sequences are reversed, proper suffixes become proper prefixes, so it follows from Lemma 1 that SpellTrieFromLeaves can be used to remove proper suffixes from a set of sequences when it is called on a trie construed with a set M_{rev} of reversed sequences, where $s \in M_{rev}$ if $s_{rev} \in M$.

When the input to Algorithm 1 is a right- supermaximal set of sequences M_{rs}, its output is a supermaximal

set of sequences M_s. Algorithm 1 removes proper suffixes from a set of sequences using SpellTrieFromLeaves and the input set M_{rs} does not have sequences that are proper prefixes or proper subsequences of other sequences (proof in next paragraph), which means that the output set M_s does not have sequences that are proper prefixes, proper subsequences, or proper suffixes of other sequences in M_s, that is, M_s is a supermaximal set of motifs.

In this paragraph we show that the input to Algorithm 1 is a right-supermaximal set of sequences. This set is produced by calling SpellTrieFromLeaves on the pruned search space trie that has all the valid motifs. It follows from the discussion in Section 2 that if a sequence is a valid motif, all its subsequences, including all its proper suffixes are valid motifs. We use this, together with Lemma 1 to show that calling SpellTrieFromLeaves produces the right-supermaximal set of motifs M_{rs}, i.e. if a motif m is in M_{rs}, any other motif in M_{rs} is neither a proper prefix (follows directly from Lemma 1) nor a proper subsequence of m. We show next that if a sequence is in M_{rs} it is not a proper subsequence of any other string in M_{rs}. Assume m is a valid motif in M_{rs}, and m_{sub} is a proper subsequence of m. m_{sub} is a proper prefix of some other sequence that is a proper suffix of m, and all proper suffixes of m are in the pruned search space trie, so it follows from Lemma 1 that m_{sub} can not be in M_{rs}.