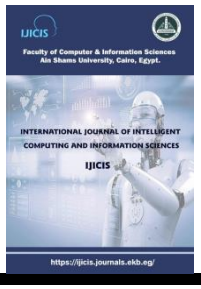
IJICIS, Vol.25, No.1, 1-17
DOI: 10.21608/ijicis.2025.358362.1375

International Journal of Intelligent Computing and Information Sciences

https://ijicis.journals.ekb.eg/



SCHEDULING MOTIF FINDING PROBLEM ACROSS HETEROGENEOUS CPU ARCHITECTURES

Ali M Elsawwaf*

Computer and Systems Engineering Dept,
Faculty of Engineering, Ain Shams University,
Cairo, Egypt.
2002336@eng.asu.edu.eg

Gamal M Aly

Computer and Systems Engineering Dept,
Faculty of Engineering, Ain Shams University,
Cairo, Egypt.
gamal.aly@eng.asu.edu.eg

Hossam M Faheem

Computer systems Department,
Faculty of Computer and Information Sciences, Ain Shams University,
Cairo, Egypt.
hmfaheem@cis.asu.edu.eg

Mahmoud Fayez

Computer systems Department,
Faculty of Computer and Information Sciences, Ain Shams University,
Cairo, Egypt.
mahmoud.fayez@cis.asu.edu.eg

Received 2025-02-05; Revised 2025-02-05; Accepted 2025-03-09

Abstract: *The Motif finding problem is important in bioinformatics, aiming to find recurring sequences in biological data. These motifs, are important for understanding how genes work, how proteins function, and how cells operate. Accurately detecting these patterns is essential for understanding of biology, aiding in scientific research, and developing treatments for diseases. Addressing the motif finding problem efficiently across heterogeneous CPU architectures presents significant challenges in computational efficiency and resource utilization. The variation in the number and speed of cores across CPUs requires developing scheduling strategies to efficiently distribute workloads among these architectures. This paper presents an efficient strategy for optimizing task distribution across heterogeneous CPU architectures. The proposed approach makes performance enhancement with 9 % solving Motif finding problem in CPU heterogeneous architectures. This improvement significantly speeds up the process of identifying important biological patterns, making bioinformatics research quicker and more cost-effective. In addition, it has a significant impact on enhancing computational efficiency and reducing costs in high-performance computing environments.*

Keywords: *Motif finding, Heterogeneous CPU architectures, Task distribution strategy*

*__Corresponding Author__: Ali M Elsawwaf

Computer and Systems Engineering Department, Faculty of Engineering, Ain Shams University, Cairo, Egypt

Email address: 2002336@eng.asu.edu.eg

1. Introduction

Motif finding problem (MFP) is a cornerstone of bioinformatics research with wide-reaching implications in understanding biological systems, elucidating disease mechanisms, and guiding therapeutic interventions. Understanding motifs can help us find out why diseases happen and how to treat them. It can even lead us to create personalized medicine, where treatments are customized for each person based on their unique genetic patterns. The motif finding problem encapsulates a significant computational challenge in bioinformatics, characterized by its inherent complexity and the vastness of biological data. At the heart of this challenge is the combinatorial explosion of possible motifs as sequence length and dataset size increase, creating a vast search space that is impractical to explore exhaustively with conventional computing resources. This complexity is compounded by the variability and degeneracy of biological motifs, where slight variations can still result in functionally equivalent motifs, further expanding the search space and complicating the identification process. Given these challenges, high-performance computing (HPC) solutions become indispensable.

HPC can handle the problem from multiple fronts:

- Offering the computational power necessary to process large datasets.
- Enabling parallel processing to explore the search space more efficiently.
- Simplifying the use of advanced algorithms that require significant computational resources. The deployment of HPC resources can dramatically reduce the time required for motif finding, making it suitable to handle the increasing scale and complexity of biological data.

This paper contributes an innovative strategy for scheduling motif finding, optimizing task distribution in heterogeneous CPU architectures that is both accurate and computationally efficient. This contribution represent significant advancement in the fields of bioinformatics and high-performance computing, offering practical solutions to some of the most pressing computational challenges.

The rest of this paper is organized as follows: Section 2 describes background of Motif finding strategies and Speed-based scheduling strategy. Section 3 presents the experimental setup, including description of the architectures used in the experiments. Section 4 presents an Exact Solution approach for solving MFP and Task Scheduling algorithm used to ensure that bioinformatics analyses are both fast and cost-effective. Section 5 presents experimental results. Finally, in Section 6 we conclude our work.

2. Background and Related Work:

This section provides an overview of the motif finding problem, including its biological importance and computational challenges. In addition, we review existing algorithms for motif finding, highlighting their limitations. This section also covers previous efforts to leverage high-performance and heterogeneous computing for bioinformatics, setting the context for the contribution of this paper.

Definition of motif finding problem: Given a set of sequences, each not necessarily has the same length, the goal is to find a motif of fixed length (usually relatively short) that occurs in each sequence of the set with few or no mismatches[1], [2], [3].

Key components of the motif finding problem:

a. Input: Set of (n) sequences $S = \{ s_1, s_2, s_3, \dots, s_n\}$ each of length (N) characters over an alphabet (A, C, G *and* T for DNA sequences or A, C, G *and* U for RNA sequences, etc.), permitted mutation, (Hamming Distance), (d) and the desired motif length (l). The motif length indicates the number of characters that the motif should consist of.
b. Task: Find motifs of length (l) that appear as a subsequence in each sequence (s_i) in the input set (S) allowing for a certain number of (d) possible mutations at maximum.
c. Output: The output of the motif finding problem is the discovered motif, which is a string of characters of the specified length (l) that appears as a subsequence in every sequence of the input set (S), allowing for a certain number of mismatches (d) at maximum.

Motif finding Algorithms and Limitations: Motif finding algorithms are key tools in bioinformatics for discovering patterns in biological sequences, essential for understanding gene regulation, protein functions, and more. However, these algorithms have several challenges:

- **Brute-Force Algorithms**[4]**:** They search all (4^l) possible sequences for motifs, becoming impractical for large datasets due to the huge number of possible motifs. This method is limited by the exponential growth in computational needs.
- **Recursive Brute-Force Algorithms**[5], [6]**:** They handle the motif finding problem by breaking down the search process into smaller, more manageable tasks. Starting with a part of the motif, these methods recursively build up to the full motif length by exploring all possible extensions one base (or amino acid) at a time. Like their brute-force counterparts, recursive brute-force algorithms are exploring all possible motifs up to the specified length. This systematic exploration can ensure high accuracy in identifying motifs. The recursive nature allows for more flexible exploration of the search space, potentially making it easier to incorporate conditions or optimizations that reduce the search space at each step. While recursive methods may offer slight improvements in managing the search space, they still face significant scalability issues due to the exponential growth of possibilities with increasing motif length and dataset size. Recursive algorithms can introduce additional computational overhead, especially if not carefully optimized. The repeated function calls and stack operations can lead to inefficiencies, particularly for deep recursion levels.
- **Greedy Algorithms**[7]**:** By iteratively building motifs and choosing the best match at each step, these algorithms can miss the best global motifs due to getting stuck in local optima, depending heavily on the starting sequence.
- **Expectation Maximization (EM) Algorithms**[8]**:** These refine motif predictions iteratively but are sensitive to initial conditions and may not find the best solution, needing substantial computational resources.
- **Gibbs Sampling**[9]**:** This probabilistic method avoids some deterministic pitfalls but can be slow to find the best motifs due to its random nature.
- **Position Weight Matrices (PWMs) and Hidden Markov Models (HMMs)**[10], [11]**:** While PWMs struggle with motifs that include variations like insertions or deletions, HMMs are complex and demand significant computational power, depending on accurate model parameters.

- **Machine Learning Approaches:** Require extensive labeled data and can be challenging to interpret, especially with complex models like neural networks.

In summary, Motif finding algorithms can be categorized into two major groups, exact and approximate solutions:

- Exact solution[5], [12], [13], [14]:
    - Apply exhaustive enumerations.
    - Guarantee global optimality.
    - Examples: Brute force, Skip Brute force, Recursive Brute force (R-BF).
- Approximate solution [7], [8], [9]:
    - Based on probability.
    - Apply some form of local search.
    - Doesn't guarantee global optimal solution.
    - Examples: Gibbs sampling and Expected maximization (EM).

A common thread among these algorithms is balancing accuracy with computation efficiency. More accurate methods (Exact Solutions) are computationally demanding, while faster ones (Approximate solutions) may lack precision. Brute-force and recursive brute-force algorithms represent foundational approaches in the quest to handle the motif finding problem, a key computational challenge in bioinformatics. These methods aim to identify recurring patterns or sequences (motifs) within a set of biological sequences by exhaustively searching through all possible sequence combinations. Despite their simplicity, these approaches have paved the way for understanding the complexities and computational demands of motif finding. Biological data's variability adds complexity, with motifs often being variable and not perfectly conserved. Addressing these issues while managing computational costs remains a key challenge, especially as data grows in size and complexity.

Parallelization efforts have tried to speed up motif finding by using multi-core CPUs, GPUs, and HPC clusters to analyze data segments concurrently, showing significant time reductions. For the purpose of enhancing motif finding and similar computational tasks, the focus is on heterogenous CPU architectures. The main idea benefits from the fact that the number of the operations required to solve Motif finding can be divided into parts. Each part can work on a specific data size called "chunk". Consequently, we can have a deterministic number of operations in almost all cases[15], [16], [17].

- **Considering task scheduling strategy**: speed-based scheduling strategy, which is displayed in figures [1], [2], is recently used in scheduling [3], [18], [19]. This strategy considers the speed of different architectures. It assumes that tasks of each chunk are executed by only a specific core related to specific architecture. This assumption eliminates the factors of sharing resources that may affect the overall system performance. Faster architecture handles a larger number of chunks. Slower architecture gets smaller number of chunks that can be exactly processed in the same time granted to the fastest architecture. Initially, execution time (T) required by the fastest architecture to handle all chunks, must be determined. Other slower architectures, that can't handle at least one chunk in the same execution time (T), are excluded as ineligible architectures.

Speed-based scheduling strategy [3], [18], [19] doesn't consider the actual execution time of a single chunk. In addition, each time we add a new architecture to the fastest one; we get a new execution time (T') of hybrid architecture smaller than (T) which should be considered on the comparison process to determine and exclude other slower architectures. This research addresses a common schedule problem, however the proposed algorithm dynamically distribute workload among different architectures. In addition, we proposed a mechanism to discard ineligible architectures efficiently.

Solving MFP needs advanced algorithms and enough computing power to search through the data efficiently, showing the vital role of high-performance computing in bioinformatics.

```
Pseudo Code for the Scheduling Strategy

Input (t1, t2, …, tp)          ; The execution times for performing the operations of a
                                        ; single chunk on A1, A2, …, Ap respectively in ascending order
Input (C)                               ; The number of chunks C = N-L+1 in case of MFP
Output (C1, C2, …, Cp)         ; The number of chunks assigned to each architecture

Function Distribute_Chunks(ti, C)

  For each i:=1 to p                    ; for each architecture satisfies this condition where
     if ((t1 * C) > ti)                         ; (2 ≤ i ≤ p) Decide which architectures will be eligible
                                                ; to perform operations on chunks
     then Ri := C / ti                  ; find the weight of each eligible architecture
     else Ri := 0                       ; or exclude ineligible architecture
  End
    R := R1 + R2 + … + Rp               ; find the total weights
    Ru := C / R                         ; find the unit assigned for each weight
  For each i:=1 to P
     Ci := Ri * Ru                              ; assign number of chunks Ci to each eligible Ai
  End
Return (C1, C2, …, Cp)
End
```

Fig. 1 Pseudo code of speed-based scheduling algorithm[18].

```
1.  PROGRAM MotifTaskScheduler
2.  Input : S[1, ..., T]
3.  Input : L
4.  Input : d
5.  BEGIN
6.  t[t1, ... ,tp] ← load single task execution time for architectures
7.  tmin ← MIN_{i=1}^{p}(ti) * 4^L   ; find the smallest run time
8.  FOR i = 1 to p
9.      IF ti ≤ tmin THEN
10.         Ri ← 4^L / t i   ; find the weight of each architecture
11.     ELSE
12.         Ri ← 0 ; this architecture is very slow and will be ignored
13.     END
14. END
15. Rtotal ← R1 + R2 + ⋯ + Rp ; sum the weights
16. Ru ← 4^L / Rtotal ; find the tasks assigned to each weight unit
    offset = 0
    starti = 0
17. FOR i = 1 to p
18.     Ci = Ri * Ru ;tasks assigned to architecture
        starti = starti + offset ; determine the start index of tasks
        endi = starti + Ci − 1 ; determine the end index of tasks
        offset = Ci
19.     Scorei
        ← SPAWN Algorithmi (S, L, d, Ci , starti , endi )
20. END
21. return MAX_{i=1}^{p} (Scorei ) ; find the motif of highest occurrence
22. END
```

Fig. 2 Pseudo code of speed-based scheduling algorithm[3], [19].

This paper has the following main objective:

Objective : Optimize Task Distribution Across Heterogeneous CPU Architectures

- Develop and implement a strategy for distributing the computational tasks involved in motif finding across a heterogeneous computing environment. This environment includes CPUs with varying numbers and speeds of cores, maximizing the utilization of available computational resources.

- Achieve Significant Reductions in Computation Time: Through development of efficient task scheduling algorithm and the efficient use of heterogeneous CPU architectures, aim to significantly reduce the overall computation time required for motif finding. This involves balancing the load across different resources applying efficient task scheduling algorithm to minimize total execution time.

3. Methodology:

This section describes in detail the methods used in the research, divided into two main parts: Considering an exact solution algorithm for motif finding and the strategy for optimizing task distribution in heterogeneous computing environments. It explains algorithmic design, and considerations for efficient computation across diverse CPU architectures.

3.1. **Considering an Exact Solution Algorithm:**

Given input set of (n) sequences $= \{ s_1, s_2, s_3, \dots, s_n\}$, each consists of 4 repeated characters $\{A, C, T, G\}$ for DNA sequences or $\{A, C, G, U\}$ for RNA sequences, not necessarily has the same length (N), the goal is to find a motif of fixed length (l) that occurs in each sequence of the set with mutation (Hamming Distance) of (d) mismatches at maximum [2].

The research represents the following exact solution algorithm:

a. Rearrange the input set of (n) sequences $S = \{ s_1, s_2, s_3, \dots, s_n\}$ in ascending order according to its length where length of $\{s_1 < s_2 < s_3 <, \dots, < s_n\}$
b. Using the shortest string (s_1) to extract set of all possible $(N - l + 1)$ windows $\{ w_1, w_2, \dots \dots, w_{N-l+1}\}$ where (N) is length of string (s_1) , and (l) is the length of the motif. These windows are used as bases for generating all possible motifs of each chunk.
c. Dividing tasks into chunks, for each window $\{ w_1, w_2, \dots \dots, w_{N-l+1}\}$ generate all possible motifs (m) $as\ in\ equation$ (1) that have mutation (Hamming Distance) of (d) mismatches at maximum with the given window. In total, we have $(N - l + 1)$ chunks, each chunk has the same total number (m) of Lmers (generated m)

$$m = \sum_{HD=0}^{d} (v-1)^{HD} C_{HD}^{l} = \sum_{HD=0}^{d} (v-1)^{HD} \frac{l!}{(l-HD)!\ HD!} \quad (1)$$

$where\ v = 4\ characters\ \{A, C, T, G\}\ for\ DNA\ sequences\ or\ \{A, C, G, U\} for\ RNA\ sequences$
$HD\ is\ hamming\ distance\ (mutation), (d)\ is\ the\ maximum\ allowed\ mutation,$
$(C)\ is\ for\ combination\ and\ (l)\ is\ length\ of\ Lmer.$

$$C_{HD}^{l} = C(l, HD) = \frac{l!}{(l-HD)!\ HD!}$$

d. For each chunk, decide if generated motif m_i of a given window is eligible. Each eligible motif must match with one window at least in each sorted sequence $\{ s_2, s_3, \dots, s_n\}$. The motif is ineligible and neglected once it has mutation greater than (d) with all extracted windows of any sequence $\{ s_2, s_3, \dots, s_n\}$ respectively.

Used exact solution algorithm for motif finding is represented in Figure (3).

1. **Start:** Begin the algorithm.
// Sort sequences by length to minimize search space.
2. **Rearrange Sequences:** Sort the input set of sequences $S = \{s_1, s_2, \ldots\ldots, s_n\}$ in ascending order based on their length.
3. **Identify Shortest Sequence:** Select the shortest string s_1 from the sorted set.
// Generate all possible motifs for the shortest sequence.
4. **Extract Windows:** Using s_1 , extract all possible windows $W = \{w_1, w_2, \ldots\ldots, W_{N-l+1}\}$, where *N* is the length of s_1 and l is the motif length.
// Process each chunk independently for parallel execution.
5. **Loop Through Chunks:** For each chunk (associated with a window from w):
- **Generate Motifs:** Generate all possible motifs m for the chunk. generate all possible motifs (m) that have up to d mismatches with each window (Hamming Distance $\leq d$).
- **Loop Through Motifs in Chunk:** For each generated motif m_i:
  - **Check Eligibility Across Sequences:** For each sequence s_j in $\{s_2, s_3, ., s_n\}$
  - **Match Motif with Windows in s_j:** Check if m_i matches at least one window in s_j with mismatches $\leq d$.
  - **Decision:** If m_i has mismatches > d with all windows of any s_j , mark m_i as ineligible.
  - **End Loop Through Motifs in Chunk**
- **End Loop Through Chunks**

6. **Collect Results:** Aggregate all eligible motifs found across all chunks.
7. **End: Conclude the algorithm.**

Fig. 3 Exact solution approach.

Task scheduling strategy:

In this modified algorithm, we consider both actual execution time of a single chunk and new execution time (T') of hybrid architecture that result each time we add a new architecture to the fastest one; where ($T' < T$). New execution time (T') is considered on the comparison process to exclude ineligible architectures as explained in figure (4) that represents the pseudo code of the proposed scheduling algorithm.

Input:
$\boldsymbol{C'}$: Predefined number of chunks
Input $(T_1, T_2, \dots\dots, T_p)$: Total execution time for a predefined number of chunks $\boldsymbol{C'}$ (respectively in ascending order) for different architectures $A_1, A_2, \dots, A_p$
where $T_1 < T_2 < \cdots < T_p$

$A_1, A_2, \dots, A_p$: Architectures $A_1, A_2, \dots, A_p$
$n_1, n_2, \dots\dots, n_p$: Number of cores for $A_1, A_2, \dots, A_p$
$t_1, t_2, \dots\dots, t_p$: Execution time for performing the operations of a single chunk on $A_1, A_2, \dots, A_p$ respectively
where $t_i = T_i/(\frac{c'}{n_i})$ and $i = 1,2, \dots\dots, p$
This doesn't mean that $(t_1 < t_2 < \cdots < t_p)$

Input(l) : Length of lmer l=15
Input(N) : Length N of shortest string s_1.
Input(C) : Total number of chunks $C = (N - l + 1)$
Output:
Output $(c_1, c_2, \dots\dots, c_p)$: Number of chunks assigned to each architecture
Output $c'_1, c'_2, \dots\dots, c'_p)$: The number of chunks per core assigned to each architecture where $c'_i = c_i/n_i$
Output (T') : Total execution Time of hybrid architecture
Begin
$R_1, R_2, \dots, R_p$: $Weight\ of A_1, A_2, \dots, A_p$
R : Total Weight
$c_1 := C$: Total chunks for A_1 is set to equal to total chunks $C = (N - l + 1)$
$\boldsymbol{c'_1 := c_1/n_1}$: The maximum assigned number of chunks per core in architecture A_1
$\boldsymbol{Remainder := c_1 \% n_1}$
If Remainder != 0
$\boldsymbol{c'_1 := [(c_1 - Remainder)\ /\ n_1] + 1}$
End
$T := t_1 * c'_1$: Minimum Total execution time of fastest architecture A_1
$T' := T$: Setting initial value of T'
$R_1 := \left(\frac{c_1}{t_1}\right) n_1$: Weight of A_1
$R := R_1$: Total Weight
$R_1, R_2, \dots, R_p := 0$: Weight of $A_1, A_2, \dots, A_p$
for each $\boldsymbol{i := 2\ to\ p}$: For each A_i decide which one is eligible
$if(T' > t_i)$
$then\ R_i := \left(\frac{C}{t_i}\right) n_i$: Get weight of eligible architecture A_i
$else\ R_i := 0$: Or exclude ineligible architecture
$R := R + R_i$: Find the total weight
$R_u := C/R$: Find the unit assigned for each weight
$offset := 0$
$start_j := 0$
for each j := 1 to p
$C_j := R_j * R_u$: Total number of assigned chunks to each eligible A_j
$start_j := start_j + offset$: Start index of chunks
$end_j := start_j + C_j - 1$: End index of chunks
$offset := C_j$
$c'_j := c_j/n_j$: Total Maximum chunks per each core in Architecture A_j
$\boldsymbol{Remainder' := c_j \% n_j}$
If Remainder' != 0
$\boldsymbol{c'_j := [(c_j - Remainder')\ /\ n_j] + 1}$
End
End
$(c_1, c_2, \dots\dots, c_p)$: Total number of assigned chunks to each eligible A_j
$(c'_1, c'_2, \dots\dots\dots, c'_p)$: Total chunks per each core in architecture A_j

$T' := t_1 * c'_1$: Updating the new total execution time of hybrid architecture that is considered
$:\simeq t_2 * c'_2$
$:\simeq t_p * c'_p$
End
End

Fig.4 Pseudo code used in proposed scheduling strategy.

4. Implementation:

In this section, we present the detailed description of experimental setup including description of CPU architectures and the total required comparison calculations. In addition, we investigate each CPU architectures to calculate the actual average execution time of a single chunk. These results serve as crucial inputs to the scheduling algorithm, which is designed to optimize resource utilization and minimize job completion times for motif finding problem.

4.1 Experimental Setup:

In this subsection, we present the detailed description of the architectures used in the experiments and required comparison calculations. In our research, we conduct experiments using a set of diverse CPU architectures.

Central Processing Units: The CPU architecture represents a traditional and versatile computing resource. We use CPUs with multiple cores and high clock speeds to handle a broad range of computational tasks, from general-purpose computing to complex simulations and data processing. Software developments, such as OpenMP and MPI (Message Passing Interface), enable bioinformatics algorithms to be parallelized and optimized for multi-core architectures. This significantly enhances the capability to process large datasets and perform complex analyses.

Description of CPU architectures:
We use 3 heterogenous CPU architectures as described in table (1). The three architectures will be denoted by Arch1, Arch2, and Arch3 respectively in the results section.

Required comparison calculations:
In this paper we consider motifs finding where $n = 20$ strings, $N = 600$ characters, $d = 4$ and $l = 15$ and $v = 4$ $characters$ $\{A, C, T, G\}$ for DNA $sequences$ of $same$ $Length$ N.
In $case$ $that$ all n $sequences$ of $strings$ $have$ the $same$ $length$ N,
$Number$ of $Windows$ per $string = Number$ of $Chunks = (N - l + 1)$
Applying equation (1), we get the total generated $Lmers$ (m) per each chunk that have $d = 4$ characters mutations at maximum.

$$Then\ Lmers\ (m) = \sum_{HD=0}^{4} (3)^{HD} C_{HD}^{15} = \sum_{HD=0}^{4} (3)^{HD} \frac{15!}{(15-HD)!\ HD!} = (\mathbf{123,841})\mathbf{Lmers/chunk} \quad (2)$$

Extracted Lmers of a given window, starting from $HD = 0,1,2,3, and\ 4\ respectively,$
$and\ l = 15$ characters per Lmer are displayed in table (2).

Table 1: Architectures (1), (2) and (3) CPU-based Compute Node.

	Architecture (1) 40 cores	Architecture (2) 24 cores	Architecture (3)4 cores
Attribute	Value	Value	Value
Architecture	x86_64	x86_64	X86-64
CPU(s)	40 vCPUs x intel® Xeon® Silver 4114	24 vCPUs Intel (R) Xeon(R) CPU E5-2630	4 vCPUs x86 Family 6 Model 15 Stepping 6 GenuineIntel
Thread(s) per core	1	2	1
Core(s) per socket	20	6	2
Socket(s)	2	2	2
CPU MHz	2.20GHZ	2.3 GHZ	2.660 GHZ
Memory	64 GB	40 GB	4 GB

Table 2: Number of extracted Lmers corresponding to each Hamming Distance (mutation).

Hamming Distance (HD)	**Number of extracted Lmers**
0	1
1	45
2	945
3	12,285
4	110,565
Total extracted Lmers per window	**123,841 Lmers**

To solve motif finding problem, we convert each string into sequences of binary bits {0,1} where $\{A, C, T\ and\ G\}$ are represented by $\{00, 01, 10\ and\ 11\}$ respectively. Accordingly, $v = 2\ bits\ \{0, 1\}$. Then each Lmer (window) has length of $l = (30)\ bits$ which is equivalent to 15 characters, and each string consists of (1200) bits which is equivalent to 600 characters.

For a given window w_i , We need to generate up to $d = 8$ bits mutations that equivalent to $d = 4$ characters mutations. Applying equation (1), we get the total generated $Lmers\ (m)$ that have up to 8 bits mutations.

$$\boldsymbol{Lmers}\ (\boldsymbol{m}) = \sum_{\boldsymbol{HD=0}}^{\boldsymbol{d}} (\boldsymbol{v-1})^{\boldsymbol{HD}} \boldsymbol{C_{HD}^{l}} = \sum_{\boldsymbol{HD=0}}^{\boldsymbol{d}} \boldsymbol{C_{HD}^{l}} = \sum_{\boldsymbol{HD=0}}^{\boldsymbol{d}} \frac{\boldsymbol{l!}}{\boldsymbol{(l-HD)!\ HD!}} = \sum_{\boldsymbol{HD=0}}^{\boldsymbol{8}} \frac{\boldsymbol{30!}}{\boldsymbol{(30-HD)!\ HD!}} \quad (3)$$

Where $v = 2\ bits\ \{0, 1\}, l = 30\ bits\ and\ d = 8$ bits.

Table (3) represents all generated Lmers applying 8 bits mutations at maximum. All generated motif generated for $d = 0$ bits mutation up to $d = 4$ bits mutations at maximum, don't need to be checked with its basis window. Accordingly, only total number of required comparisons will follow equation (4)

$$\boldsymbol{Required\ Comparisons} = \sum_{\boldsymbol{HD=5}}^{\boldsymbol{8}} \boldsymbol{C_{HD}^{30}} \quad (4)$$

These comparisons are applied with each $(N - l + 1)$ window to get a list of (123,841) Lmers that have total of 4 characters mutations at maximum for a given window, w_i that are displayed in table (2).

Accordingly, total number of comparisons calculations =

= (Number of strings * Number of windows per string* Number of chunks* Number of generated Lmers per chunk) + (Number of chunks* Required comparisons of each chunk)

$$= \left[(n-1) * (N-l+1)^2 * \sum_{HD=0}^{4} (3)^{HD} C_{HD}^{15}\right] + \left[(N-l+1) * \sum_{HD=5}^{8} C_{HD}^{30}\right] \quad (5)$$

Considering that all calculations of a single chunk, are carried by only a specific core inside the given architecture, including creation of motifs and comparison operations with all extracted Lmers of given sequence of strings are displayed in equation (6) to investigate all accepted Lmers.

$$\text{Total calculations of a single chunk} = \left[(n-1)*(N-l+1)*\sum_{HD=0}^{4}(3)^{HD}C_{HD}^{15}\right]+\left[\sum_{HD=5}^{8}C_{HD}^{30}\right] \quad (6)$$

Table 3: Number of extracted Lmers corresponding up to 8 bits mutations at maximum.

HD ($bits$)	l ($bits$)	$l-HD$	$l!$	$HD!$	$(l-HD)!$	$Lmers$ (m)
0	30	30	2.65253E+32	1	2.65253E+32	1
1	30	29	2.65253E+32	1	8.84176E+30	30
2	30	28	2.65253E+32	2	3.04888E+29	435
3	30	27	2.65253E+32	6	1.08889E+28	4,060
4	30	26	2.65253E+32	24	4.03291E+26	27,405
5	30	25	2.65253E+32	120	1.55112E+25	142,506
6	30	24	2.65253E+32	720	6.20448E+23	593,775
7	30	23	2.65253E+32	5040	2.5852E+22	2,035,800
8	30	22	2.65253E+32	40320	1.124E+21	5,852,925
Total Extracted Lmers (up to 8 bits mutations at maximum) per window						**8,656,937**

4.2. Calculating the Actual Average Execution Time of a Single Chunk:

We apply two different solvers to extract motifs. In the first solver we use recursive function to generate motifs of each chunk, while the second solver replace recursive function with iterative one. For both solvers, we use MPI paradigm. We start by investigating each CPU architecture to calculate the actual average execution time of a single chunk. These results serve as inputs to the proposed scheduling algorithm, which is designed to optimize resource utilization and minimize job completion times for large-scale problems. We have a total chunks of $(N-l+1)=586$.

Firstly, we investigate execution time for each chunk in case of activating only 1 core in each architecture. Secondly, we reinvestigate execution time for each chunk in case of activating all cores in each architecture.

Figures [5] display average execution time of a single chunk in different CPU architectures, applying different number of cores and two different solvers. In case of applying solver1 (recursive one), average execution time of a single chunk is represented in blue bar, while it is represented in black bar when applying solver2 (iterative one) for each architecture. Investigating figure [5], solver2 (iterative one) outperforms solver1 (recursive one), Accordingly, solver2 is used to get better execution time. Figure [6] display average execution time for a single chunk in both cases of activating only one core and activating all cores of each architecture.

Investigating figures [5-6], we conclude that:

1. For all architectures, average execution time of a chunk using the second solver (iterative one) outperforms that of the first solver (recursive one).

2. For all architectures, average execution time of each chunk increases as the number of cores increase whatever which solver is used due to communication overhead between master core and all remaining cores. We get the maximum execution time of a single chunk, displayed in black line, when all cores of a given architecture are used in the computational process.

Accordingly:

1. We apply the best problem solver (iterative one) to get best execution time.
2. To get actual execution time of a single chunk, we consider the architecture that has maximum number of cores. Accordingly, we set the predefined number of chunks ($\boldsymbol{C' = 40}$) to ensure that all cores of each architecture are used and activated.

As a result, we get actual average execution time of each chunk applying the best solver (iterative one) of each architecture as displayed in figure (7). To get average total execution time of all chunks, we multiply average execution time of a chunk by the maximum number of chunks assigned to single core for each architecture.
Table (4) represents both of actual total execution time of all 586 chunks and average execution time of a single chunk (activating all cores) for different architectures. These results are displayed in figures (7, 8).

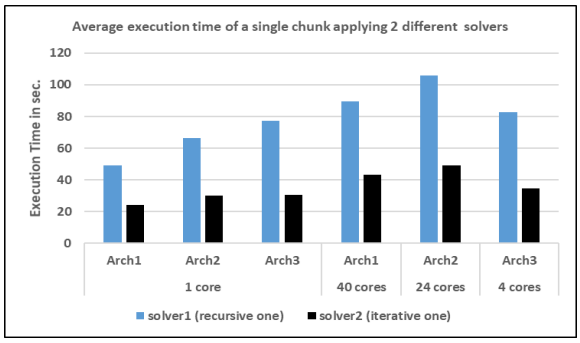


Fig.5 Average Execution time of each chunk for all Architectures, applying two different solvers.

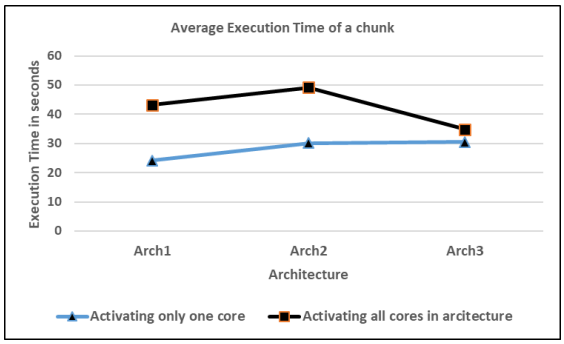


Fig.6 Execution time of a single chunk, applying solver2 (iterative one).

Table 4: Total execution time and actual average execution time (Sec.) of single chunk for different architectures.

#Chunks	**Architecture**		
	Arch. (1), 40 cores, 2.20GHZ	**Arch. (2), 24 cores, 2.3 GHZ**	**Arch. (3), 4 cores, 2660 Mhz**
$N - l + 1 = 586$	661.0124	1242.3385	5191.6137
1	43.21405	49.14978	34.82841

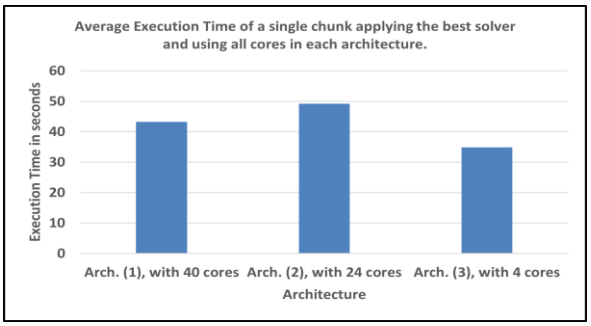


Fig.7 Average execution time of single chunk, activating all cores in each architecture.

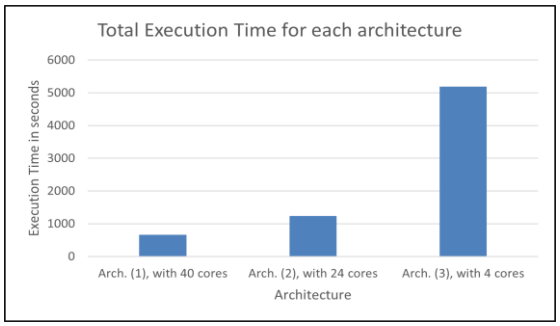


Fig.8 Actual total execution time of all 586 chunks for different architectures.

5. Experimental Results:

This section presents the findings of the study, starting with the validation of the exact solution algorithm's accuracy and efficiency. It then evaluates the effectiveness of the task distribution strategy, demonstrating its impact on computation time and resource utilization. The results are compared with existing methods to highlight the improvements achieved.

5.1. Problem Complexity Reduction:

1- Applying Skip Brute Force[20]:
For a given length (15) of L-mer, we get the total number of 4^{15} L-mers that need to be investigated with each window in the first string to extract all accepted (123,841) L-mers that will need to be investigated with $(n-1)(N-l+1)$ windows where n is the number of strings, N is the length of each string and l is the length of L-mer.

2- Applying used algorithm:
Instead of investigating all of 4^l L-mers as in case of skip brute force with each window in the first string, we only need to investigate a specific number of Lmers presented in equation (4) applying 8 bits binary bit mutations at maximum for each window as displayed in table (3).
Accordingly, we get a reduction in total number of L-mers that need to be investigated.

$$\textbf{Percentage of Total Reduction} = \frac{4^{15} - \sum_{HD=5}^{8} C_{HD}^{30}}{4^{15}} = 99.2\% \quad (7)$$

5.2. Applying Proposed Scheduling algorithm:

A- The proposed scheduling algorithm considers actual execution time of single chunk while activating all cores of each architecture in the computation process. Using results in table (4) and applying scheduling algorithm in figure (4), total number of 586 chunks are divided between different architectures as displayed in table (5). Combining these architectures result in a hybrid architecture that consists of 68 cores (ranks) that take index from 0 to 67 and start its work in parallel.

Total execution time of the hybrid architecture is the execution time of the latest rank finishing its work.

B- Speed-based scheduling strategy, which is displayed in figures [1-2], are recently used in scheduling [3], [18], [19]. This algorithm doesn't consider the new execution time (T') of hybrid architecture to exclude illegible architectures. In addition, This algorithm activates only one core to handle all computations for this process. Accordingly, execution time of single chunk, activating only one core, is taken as a reference for each architecture which is represented in blue line in figure [6]. Accordingly, speed-based scheduling algorithm [3], [18], [19] doesn't consider the actual execution time of single chunk that is represented in black line in figure [6]. Tables (5, 6) represent a comparison of assigned chunks for each architecture and total execution time between scheduling algorithm [3], [18], [19] and the proposed approach.

Table 5: Chunk assignment based on different scheduling algorithm.

Scheduling Algorithm	**Architecture**		
	Arch. (1), 40 cores, 2.20GHZ	**Arch. (2), 24 cores, 2.3 GHZ**	**Arch. (3), 4 cores, 2660 Mhz**
Proposed approach	355 chunks	187 chunks	44 chunks
Algorithm [3], [18], [19]	376 chunks	180 chunks	30 chunks

Table 6: Total execution time for algorithm [3], [18], [19] and new proposed approach.

Total Execution Time in Seconds		
Scheduling Algorithm [3], [18], [19]	**New Proposed Approach**	**Enhancement %**
4.3214E+02	3.9320E+02	9%

Performance Enhancement $= \frac{T_{old}-T_{new}}{T_{old}} = 9\%$

6. Conclusion:

In this study, we proposed scheduling strategy to optimize resource utilization and minimize job completion times for large-scale problems across heterogeneous CPU architectures. The research aimed to address the challenges associated with CPU heterogeneous architectures, leading to inefficient resource utilization and extended job completion times. To achieve our objectives, we first conducted a series of experiments using various CPU architectures and initiating different number of cores to calculate actual execution time of single chunk. Based on these findings, we developed the proposed scheduling strategy that maps computation of chunks to architectures based on their actual execution time of single chunk and new total execution time (T') of hybrid eligible architectures. While scheduling algorithm[3], [18], [19] uses total execution time (T) of fastest architecture as a static reference for comparison process to eliminate ineligible architecture, we assure that the new resulting actual total execution time (T') of the hybrid eligible architecture should be used in the comparison process instead. In addition, the actual execution time for a single chunk should be considered for each architecture. We examine the performance trends of each architecture applying different numbr of cores. This analysis helps us understand how each architecture scales with increasing number of cores. Identifying the performance trends allows us to determine the architecture's efficiency in handling a

wide range of computational workloads. Evaluation of the proposed approach demonstrated several key findings. First, the proposed scheduling approach significantly improved resource utilization compared to scheduling algorithm[3], [18], [19]. Second, the proposed scheduling algorithm led to faster job completion and improved system efficiency by 9%. Future research directions are proposed to improve the architecture-aware scheduling approach further. These directions include exploring energy-aware scheduling into the scheduling approach. Energy-efficient scheduling aims to minimize energy consumption while maintaining high-performance levels. By incorporating energy-awareness, the approach can contribute to more sustainable and environmentally friendly computing practices.

7. Declaration

Competing interests

The author(s) declare no competing interests.
On behalf of all authors, the corresponding author states that there is no conflict of interest.

Data availability

Data is provided within the manuscript

8. References

[1] P. Pevzner, *Computational molecular biology: an algorithmic approach*. MIT press, 2000.

[2] F. A. Hashim, M. S. Mabrouk, and W. Al-Atabany, "Review of different sequence motif finding algorithms," *Avicenna J Med Biotechnol*, vol. 11, no. 2, p. 130, 2019.

[3] A. Barghash and A. Harbaoui, "A Proposed Approach for Motif Finding Problem Solved on Heterogeneous Cluster with Best Scheduling Algorithm," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 5, 2023.

[4] H. M. Faheem, "Accelerating Motif Finding Problem using Grid Computing with Enhanced Brute Force," 2010.

[5] M. A. Radad, N. A. El-Fishawy, and H. M. Faheem, "Implementation of Recursive Brute Force for Solving Motif Finding Problem on Multi-Core," *International Journal of Systems Biology and Biomedical Technologies*, vol. 2, no. 3, pp. 1–18, Jul. 2013, doi: 10.4018/ijsbbt.2013070101.

[6] M. A. Radad, N. A. El-Fishawy, and H. M. Faheem, "Enhancing Parallel Recursive Brute Force Algorithm for Motif Finding," 2014.

[7] A. M. Carvalho and A. L. Oliveira, "GRISOTTO: A Greedy Approach to Improve Combinatorial Algorithms for Motif Discovery with Prior Knowledge," *Algorithms for Molecular Biology*, vol. 6, 2011, [Online]. Available: https://link.springer.com/article/10.1186/1748-7188-6-13

[8] J. M. C. Garbelini and D. S. Sanches, "Expectation Maximization Based Algorithm Applied to DNA Sequence Motif Finder," in *2022 IEEE Congress on Evolutionary Computation*, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9870303/

[9] G. D. Stormo, *Motif Discovery Using Expectation Maximization and Gibbs Sampling*. Springer, 2010. [Online]. Available: https://link.springer.com/protocol/10.1007/978-1-60761-854-6_6

[10] M. Siebert and J. Söding, “Bayesian Markov models consistently outperform PWMs at predicting motifs in nucleotide sequences,” *Nucleic Acids Res*, vol. 44, no. 13, pp. 6055–6069, 2016.

[11] J. Wu and J. Xie, “Hidden Markov model and its applications in motif findings,” *Statistical Methods in Molecular Biology*, pp. 405–416, 2010.

[12] M. Hasan and P. C. Shill, “A Comparative Analysis for Generating Common d-Neighborhood on Planted Motif Search Problem,” in *International Conference on Intelligent Computing & Optimization*, 2022, pp. 822–831.

[13] S. Mohanty, P. K. Pattnaik, A. A. Al-Absi, and D.-K. Kang, “A Review on Planted (l, d) Motif Discovery Algorithms for Medical Diagnose,” *Sensors*, vol. 22, no. 3, p. 1204, 2022.

[14] M. Hasan, A. S. M. Miah, M. M. Hossain, and M. S. Hossain, “LL-PMS8: A time efficient approach to solve planted motif search problem,” *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 6, pp. 3843–3850, 2022.

[15] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, “Portable performance on heterogeneous architectures,” in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2013, pp. 431–443. doi: 10.1145/2451116.2451162.

[16] A. R. Arunarani, D. Manjula, and V. Sugumaran, “Task scheduling techniques in cloud computing: A literature survey,” *Future Generation Computer Systems*, vol. 91, pp. 407–415, Feb. 2019, doi: 10.1016/j.future.2018.09.014.

[17] H. M. Faheem and B. König-Ries, “A multiagent-based framework for solving computationally intensive problems on heterogeneous architectures bioinformatics algorithms as a case study,” in *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems*, SciTePress, 2014, pp. 526–533. doi: 10.5220/0004967105260533.

[18] H. M. Faheem and others, “A new scheduling strategy for solving the motif finding problem on heterogeneous architectures,” *Int J Comput Appl*, vol. 101, no. 5, 2014.

[19] H. M. Faheem, B. Koenig-Riez, M. Fayez, I. Katib, and N. AlJohani, “Solving the Motif Finding Problem on a Heterogeneous Cluster using CPUs, GPUs, and MIC Architectures,” *Mathematics and Computers in Sciences and Industry*, pp. 226–232, 2015.

[20] M. M. Al-Qutt, H. Khaled, R. ElGohary, H. M. Faheem, I. Katib, and N. Al-Johani, “Accelerating Motif Finding Problem Using Skip Brute-Force on CPUs and GPU’s Architectures,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2017, pp. 155–161.