Journal of Computing and Security

Research Article, 2026, Volume 13, Number 2 (pp. 49–61)

http://www.jcomsec.org

Enhancing Error Detection Performance through Parallel CRC Computation on Multi-Core Architectures

Mohammadjavad Khani [a] Mahmood Ahmadi [a,*]

[a] *Computer Engineering and Information Technology Department, Razi University, Kermanshah, Iran.*

ARTICLE INFO.

Article history:
Received: 27 February 2026
Revised: 02 July 2026
Accepted: 27 July 2026
Published Online: 20 August 2026

Keywords:
Cyclic Redundancy Check (CRC), Parallel Computing, Multithreading, Energy Efficiency, Performance Evaluation.

ABSTRACT

Cyclic Redundancy Check (CRC) remains one of the most widely used error-detection mechanisms in communication, storage, and embedded systems. However, conventional software CRC implementations suffer from inherent sequential dependencies that limit efficient utilization of modern multi-core processors. This paper presents a generalized software-based parallel CRC framework for multi-core architectures using POSIX threads (pthreads). The proposed framework supports multiple CRC variants, including CRC-8, CRC-16, CRC-32, CRC-64, and CRC-128, within a unified implementation model. To preserve correctness during parallel execution, the framework employs a GF(2)-based CRC combination mechanism rather than naïve XOR aggregation. The combine stage is formulated using polynomial arithmetic and matrix-based shifting operations over GF(2), ensuring equivalence between parallel and serial CRC computation. The proposed method was evaluated using multiple workload sizes and thread configurations. Experimental analysis includes execution time, throughput, latency, scalability behavior, and energy estimation under varying thread counts. Results indicate that parallel execution significantly improves performance for large datasets, achieving approximately 3–4x speedup on the evaluated platform while preserving exact CRC correctness. Comparative discussion with representative CRC optimization approaches, including lookup-table methods, slicing-by-8, SIMD/vectorized CRC, and hardware-assisted CRC techniques, is also provided to position the proposed framework within the broader CRC optimization landscape. Although the presented framework emphasizes portability, multi-variant support, and correctness-preserving software parallelization, scalability remains influenced by synchronization overhead, memory bandwidth constraints, and hardware characteristics. Overall, the proposed approach provides a portable and generalized software framework for correctness-preserving parallel CRC acceleration on general-purpose multi-core systems.

1 Introduction

Reliable error detection remains a fundamental requirement in modern digital communication, storage, networking, and embedded computing systems. During transmission or storage, data may be corrupted by noise, hardware faults, electromagnetic interference, or synchronization errors. Consequently, efficient mechanisms for detecting accidental bit modifications are essential for maintaining data integrity.

Among various error-detection techniques, Cyclic Redundancy Check (CRC) has become one of the most widely adopted methods because of its strong error-detection capability, low implementation complexity, and flexibility across different application domains [1], [2]. CRC mechanisms are extensively used in communication protocols, storage devices, Ethernet networks, wireless systems, embedded controllers, and file integrity verification frameworks [3].

CRC computation is traditionally implemented through polynomial arithmetic over the finite field GF(2) [1], [2]. Although CRC algorithms are computationally efficient, conventional software implementations exhibit an inherent sequential dependency. The CRC state generated for a given data element depends directly on the previously computed state, creating a recursive processing chain. As a result, straightforward exploitation of modern multi-core processors becomes challenging.

The increasing prevalence of multi-core and many-core computing platforms motivates renewed interest in software parallelization strategies for CRC computation. Contemporary processors provide substantial thread-level parallelism capabilities that can potentially accelerate CRC workloads for large datasets. However, achieving correct CRC parallelization is non-trivial. A naïve division of the input stream followed by direct XOR aggregation of partial CRC values does not preserve the mathematical properties of CRC concatenation because CRC results depend on message ordering and polynomial state advancement [4], [5].

Several optimization techniques have been proposed to improve CRC performance, including lookup-table approaches, slicing-by-8 methods, SIMD/vectorized implementations, carry-less multiplication techniques, and hardware-assisted CRC instructions [6], [7], [8]. While these approaches provide substantial acceleration in specific environments, they frequently focus on particular CRC variants, hardware features, or instruction-set support.

In addition, many existing software studies emphasize CRC-32 exclusively, leaving broader multi-variant implementations comparatively underexplored [8], [9]. Furthermore, practical considerations such as portability, correctness-preserving combine operations, scalability behavior, and energy-oriented evaluation remain important for software implementations intended for general-purpose processors.

This paper presents a generalized software-based framework for parallel CRC computation using POSIX threads (pthreads). The proposed approach supports multiple CRC families, including CRC-8, CRC-16, CRC-32, CRC-64, and CRC-128, under a unified implementation model. To preserve mathematical correctness during parallel execution, the framework employs a GF(2)-based CRC combination mechanism derived from polynomial arithmetic and matrix-based shifting operations rather than naïve XOR aggregation [10], [11].

The proposed framework is evaluated using execution time, throughput, latency, scalability behavior, and energy-oriented measurements under different workload sizes and thread configurations. The study additionally discusses synchronization overhead, memory bandwidth effects, and practical scalability limitations associated with software CRC acceleration on multi-core systems.

The main contributions of this work can be summarized as follows:

(1) A generalized pthread-based software framework supporting multiple CRC variants, including CRC-8, CRC-16, CRC-32, CRC-64, and CRC-128.
(2) A correctness-preserving GF(2)-based CRC combination mechanism for parallel chunk aggregation.
(3) Extended experimental evaluation incorporating throughput, latency, scalability analysis, and energy-oriented measurements.
(4) Discussion of practical implementation trade-offs involving synchronization overhead, memory behavior, and multi-core scalability considerations.

The remainder of this paper is organized as follows. Section 2 reviews and analyzes existing and related works. Section 3 introduces the mathematical background and implementation methodology. Section 4 presents the proposed parallel CRC framework and GF(2)-based combine mechanism. Section 5 reports experimental evaluation and discussion. Finally, Section 6 concludes the paper and outlines future research

* Corresponding author.
Email addresses: m.khani@razi.ac.ir (M. Khani), m.ahmadi@razi.ac.ir (M. Ahmadi)
https://dx.doi.org/10.22108/jcs.2026.148544.1198
ISSN: 2322-4460

directions.

2 Related Works

CRC optimization has been extensively studied across software, hardware, networking, storage, and embedded computing environments because of the widespread use of CRC in error detection applications [1], [12], [2].

Early software CRC implementations primarily relied on bitwise polynomial division methods [1]. Although conceptually simple and mathematically straightforward, bitwise implementations often suffer from relatively high computational cost because each input bit must be processed sequentially through repeated shift and XOR operations.

To improve software performance, lookup-table approaches were introduced [7]. Table-driven CRC methods precompute polynomial remainder values to reduce runtime computational overhead. Variants such as byte-wise lookup and slicing-by-4/slicing-by-8 techniques significantly accelerate CRC processing by exploiting precomputed transformation tables and wider memory accesses [7], [8]. These approaches remain widely used in practical software libraries because of their favorable balance between simplicity and performance.

Further acceleration has been achieved through instruction-level optimization methods. SIMD (Single Instruction Multiple Data) implementations leverage vector processing instructions such as SSE and AVX to process multiple data elements concurrently. In addition, carry-less multiplication techniques and dedicated hardware CRC instructions available on modern processors have demonstrated substantial performance improvements for selected CRC variants, particularly CRC-32 [6], [8]. However, such approaches often depend on specific processor architectures, instruction-set extensions, or hardware capabilities.

Several software libraries and operating system kernels incorporate optimized CRC combination mechanisms for concatenated data processing. For example, GF(2)-based combine operations have been employed in implementations such as crc32-combine to correctly merge CRC states derived from segmented message processing [10], [11]. These approaches recognize that CRC values cannot be combined through naïve XOR aggregation because CRC correctness depends on polynomial state advancement and message ordering [7].

Parallel CRC computation has also received attention in prior literature. Existing studies have explored multi-threaded CPU implementations, FPGA-based architectures, and specialized hardware pipelines to increase throughput for large-scale datasets and high-speed communication environments [13], [14], [15], [16], [9]. Hardware-oriented solutions frequently achieve very high acceleration; however, they may require specialized development environments or reduced portability compared with general-purpose software implementations.

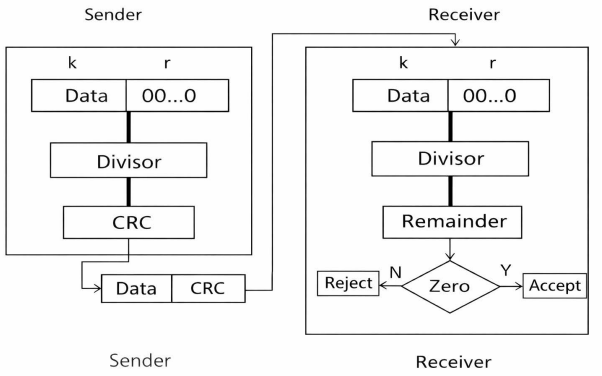


Figure 1. Block Diagram of cyclic redundancy check [21].

Despite significant prior progress, several practical gaps remain. Many published studies concentrate primarily on CRC-32 and do not examine a broader family of CRC variants within a unified framework [8], [9]. Furthermore, some approaches prioritize raw throughput optimization while providing limited discussion of correctness-preserving combination mechanisms, scalability behavior, synchronization overhead, portability considerations, or energy-oriented evaluation.

In contrast to hardware-specific acceleration methods, the present work focuses on a generalized software-oriented approach for multi-core processors using POSIX threads (pthreads). The proposed framework supports multiple CRC variants—including CRC-8, CRC-16, CRC-32, CRC-64, and CRC-128—within a unified implementation structure. A GF(2)-based combine formulation is employed to preserve equivalence between serial and parallel execution [10], [11], and the experimental evaluation incorporates execution time, throughput, latency, scalability behavior, and energy-oriented analysis.

Table 1 summarizes representative categories of existing CRC optimization approaches and positions the proposed framework relative to prior work. As shown in Table 1, the proposed framework is not intended to compete directly with highly specialized hardware-assisted CRC engines. Instead, it emphasizes portability, generalized multi-variant support, correctness-preserving parallel combination, and software-oriented evaluation on general-purpose multi-core platforms.

Table 1. Representative CRC Optimization Approaches

Approach	Software	Parallel	Multi-CRC Support	Hardware Dependency	Correct Combine
Bitwise CRC	Yes	No	Partial	No	Yes
Lookup-Table CRC	Yes	No	Partial	No	Yes
Slicing-by-8	Yes	Limited	Mainly CRC-32	No	Yes
SIMD / Vectorized CRC	Yes	Yes	Limited	Processor-Specific	Yes
Hardware CRC Instructions	Partial	Yes	Limited	High	Yes
FPGA / Hardware CRC	No	Yes	Variable	High	Yes
Proposed Framework	**Yes**	**Yes**	**Yes**	**No**	**Yes**

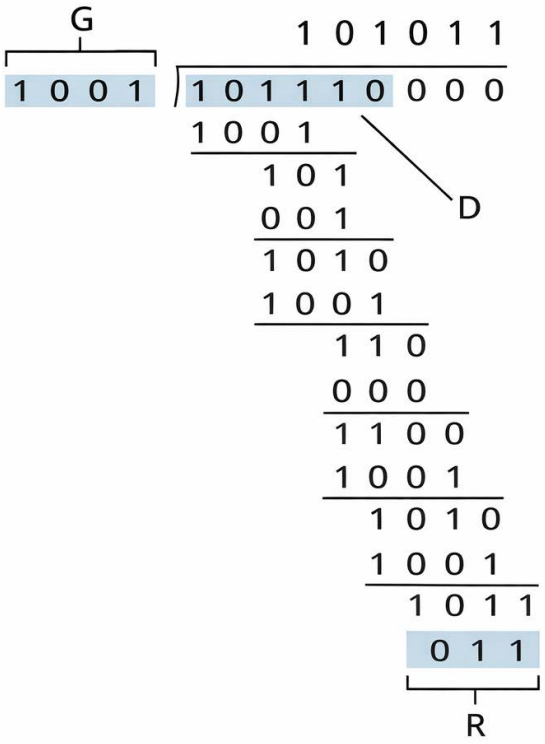


Figure 2. A sample CRC calculation [20].

3 Fundamentals of Cyclic Redundancy Check (CRC)

Cyclic Redundancy Check (CRC) is a widely used error detection mechanism designed to identify accidental changes in digital data during transmission or storage. CRC is based on polynomial arithmetic over the Galois Field GF(2), where both the data stream and the generator polynomial are represented as binary polynomials. The CRC value is computed by appending a fixed number of zero bits to the original data and performing a polynomial division by a predefined generator polynomial as shown in Fig. 1. The remainder of this division constitutes the CRC checksum, which is appended to the transmitted data. At the receiver side, the same division is performed to verify data integrity. As shown in Fig. 2, a sample CRC calculation is illustrated. The strength of CRC lies in its ability to detect common error patterns such as single-bit errors, burst errors, and multiple-bit errors, depending on the choice of the generator polynomial. Due to its strong detection capability and low redundancy overhead, CRC has been adopted as a standard error detection technique in numerous communication protocols and storage systems [1], [3], [2].

From an implementation perspective, CRC can be realized in both hardware and software, each offering distinct advantages. Hardware implementations often rely on linear feedback shift registers (LFSRs), enabling high-speed and low-latency CRC computation, which is particularly suitable for network interfaces and embedded controllers [13], [17], [18]. Software-based CRC implementations, on the other hand, provide greater flexibility and portability across different platforms. In software, CRC is commonly implemented using bitwise operations, lookup tables, or a combination of both. Bitwise implementations are simple and memory-efficient but tend to be computationally expensive, whereas table-driven implementations trade memory usage for higher throughput by precomputing partial CRC values [7], [10], [11]. The choice of implementation strategy depends on system constraints such as available memory, required throughput, and power consumption.

Performance considerations play a crucial role in CRC computation, especially in high-speed and data-intensive applications. The execution time of CRC algorithms is influenced by factors such as input data size, processor architecture, memory access patterns, and instruction-level parallelism. On modern processors, cache efficiency and branch prediction can significantly affect CRC performance. While table-based approaches reduce computational complexity, they may suffer from cache misses when large tables are used. Conversely, bitwise implementations are compute-bound and may limit throughput on large data streams. As data rates and data volumes continue to increase, optimizing CRC performance has become

a critical challenge in software systems, particularly those operating under real-time or energy-constrained conditions [6], [7], [8], [12].

With the widespread adoption of multi-core processors, parallel computing has emerged as an effective solution to overcome the performance limitations of traditional serial CRC implementations. Parallel CRC techniques divide input data into independent segments that can be processed concurrently, thereby exploiting thread-level parallelism to improve computational efficiency [13], [4], [5], [15], [16], [9]. However, extending parallelization beyond a single CRC type introduces additional challenges, as different CRC variants (e.g., CRC-8, CRC-16, CRC-32, CRC-64, and CRC-128) exhibit distinct polynomial structures and computational complexities. Efficient parallel designs must therefore address workload balancing, synchronization overhead, and the correct combination of partial CRC results while preserving algorithmic correctness across multiple CRC formats [4, 5, 9, 19]. A comprehensive understanding of the computational characteristics and structural differences among various CRC algorithms is thus essential for motivating, designing, and systematically evaluating the proposed parallel CRC framework presented in this paper.

4 Proposed Parallel CRC Framework

This section presents the proposed generalized software framework for parallel CRC computation on multi-core processors. The framework combines thread-level parallelism with a mathematically correct GF(2)-based combination mechanism to preserve equivalence with serial CRC execution.

4.1 Threading Model and Static Data Partitioning

The proposed implementation employs POSIX threads (pthreads) because of their portability, deterministic synchronization behavior, and widespread availability across general-purpose operating systems. Given an input dataset D of length L bytes and a thread count T, the input message is partitioned into contiguous chunks:

$$D = D_0||D_1||D_2||\cdots||D_{T-1} \tag{1}$$

where:

$$|D_i| \approx \frac{L}{T} \tag{2}$$

Each thread independently computes the CRC value of its assigned partition.

Static partitioning is adopted to reduce runtime scheduling overhead, improve cache locality, and maintain predictable workload distribution. When the input size is not divisible by the number of threads, the remaining bytes are assigned to the final thread.

The chunk boundaries are computed as:

$$Start_t = t \times \left\lfloor \frac{L}{T} \right\rfloor \tag{3}$$

$$End_t = \begin{cases} L, & t = T-1 \\ Start_t + \left\lfloor \frac{L}{T} \right\rfloor, & otherwise \end{cases} \tag{4}$$

This strategy avoids synchronization during local CRC computation and introduces synchronization only during thread joining and result combination.

4.2 Mathematical Formulation of CRC Combination

CRC computation can be represented through polynomial arithmetic over the finite field GF(2). Let:

$$P(x) \tag{5}$$

denote the generator polynomial. Assume an input message consisting of two concatenated segments:

$$M = A||B \tag{6}$$

where A and B represent message chunks. A common misconception is to combine partial CRC results using simple XOR aggregation:

$$CRC(A||B) \neq CRC(A) \oplus CRC(B) \tag{7}$$

because CRC correctness depends on message ordering and polynomial state advancement. Instead, CRC linearity yields:

$$CRC(A||B) = (CRC(A) \cdot x^{8|B|} \bmod P(x)) \oplus CRC(B) \tag{8}$$

where $|B|$ denotes the byte length of the second segment. The term

$$x^{8|B|} \tag{9}$$

represents advancement of the CRC register state by the number of bits contained in the appended message segment.

A generalized combine operator can therefore be defined as:

$$Combine(c_1, c_2, l_2) = (c_1 \otimes Shift(l_2)) \oplus c_2 \tag{10}$$

where:

- c_1 = CRC of the first chunk,
- c_2 = CRC of the second chunk,
- $Shift(l_2)$ = GF(2)-based advancement operator.

This formulation guarantees equivalence between segmented parallel execution and serial CRC computation.

4.3 GF(2)-Based Matrix Combination Mechanism

Direct evaluation of polynomial advancement through repeated multiplication is computationally expensive for large message sizes. To improve efficiency, the proposed framework employs a matrix-based shifting mechanism over GF(2).

Let:

$$M \tag{11}$$

denote the CRC transition matrix. Advancing the CRC state by one bit is represented as:

$$v' = Mv \tag{12}$$

where v denotes the CRC register state vector.

Advancement by k bits becomes:

$$v^{(k)} = M^k v \tag{13}$$

The exponentiation stage is efficiently computed using repeated squaring:

$$M^k = \prod_i M^{2^i} \tag{14}$$

which reduces the computational cost from linear repeated shifting to logarithmic-exponentiation complexity.

The matrix transformation is adapted to match the processing convention of each CRC family:

- CRC-8, CRC-16, CRC-64: MSB-first representation.
- CRC-32: reflected implementation.
- CRC-128: extended 128-bit register representation.

Consequently, the combine stage remains mathematically valid across multiple CRC variants.

4.4 Correctness Analysis

Theorem 1.

The proposed GF(2)-based combination procedure produces a CRC value identical to serial CRC computation over the original concatenated message.

Proof.

CRC computation constitutes a linear transformation over GF(2). For a message:

$$M = A||B \tag{15}$$

serial CRC execution is equivalent to:

(1) processing segment A,
(2) advancing the resulting state by $|B|$ bytes,
(3) incorporating the CRC contribution of B.

Algorithm 1 Generalized Parallel CRC Framework

Require: Dataset D, length L, thread count T, CRC type
Ensure: Final CRC value
$chunk\_size \leftarrow \lfloor L/T \rfloor$
for $t = 0$ to $T-1$ **do**
  $Start[t] \leftarrow t \times chunk\_size$
  if $t = T-1$ **then**
    $End[t] \leftarrow L$
  else
    $End[t] \leftarrow Start[t] + chunk\_size$
  end if
end for
for all threads t in parallel **do**
  $CRC_{partial}[t] \leftarrow SerialCRC(D[Start[t] : End[t]])$
end for
Wait for all threads to complete
$CRC_{final} \leftarrow CRC_{partial}[0]$
for $t = 1$ to $T-1$ **do**
  $CRC_{final} \leftarrow$
    $GF2Combine(CRC_{final},$
    $CRC_{partial}[t],$
    $chunk\_length[t])$
end for
return CRC_{final}

Algorithm 2 GF(2)-Based CRC Combination Procedure

Require: crc_1, crc_2, len_2, generator polynomial $P(x)$
Ensure: Combined CRC value
if $len_2 = 0$ **then**
  return crc_1
end if
Construct the GF(2) transition matrix M
Compute the shift matrix
  $S \leftarrow M^{8 \times len_2}$
Apply state advancement
  $shifted\_crc \leftarrow Apply(S, crc_1)$
Combine shifted state with second CRC
  $combined\_crc \leftarrow shifted\_crc \oplus crc_2$
return $combined\_crc$

Using Eq.(8):

$$CRC(A||B) = (CRC(A) \cdot x^{8|B|} \bmod P(x)) \oplus CRC(B) \tag{16}$$

which matches the algebraic formulation implemented by the proposed combine procedure.

Therefore:

$$CRC_{parallel} = CRC_{serial} \tag{17}$$

and correctness is preserved.

4.5 Complexity Analysis

Let:

- N = input size,
- T = thread count.

The serial CRC implementation processes the complete dataset sequentially:

$$O(N) \tag{18}$$

Under parallel execution, each worker thread processes approximately:

$$N/T \tag{19}$$

bytes.

Therefore, chunk computation complexity becomes:

$$O(N/T) \tag{20}$$

The combine stage employs logarithmic matrix exponentiation:

$$O(\log L) \tag{21}$$

where L denotes chunk length.

Overall complexity is therefore:

$$O(N/T) + O(\log L) \tag{22}$$

For sufficiently large datasets:

$$\log L \ll N/T \tag{23}$$

and runtime is dominated primarily by parallel chunk computation.

Synchronization overhead is intentionally minimized because threads exchange neither intermediate states nor shared buffers during local processing. Nevertheless, scalability may be limited by:

- synchronization cost,
- cache contention,
- memory bandwidth saturation,
- Hyper-Threading resource sharing.

Such behavior is consistent with Amdahl's Law:

$$S(T) = \frac{1}{(1-P) + \frac{P}{T}} \tag{24}$$

where P denotes the parallelizable workload fraction.

4.6 CRC-128 Implementation Details

Unlike CRC-32 and CRC-64, CRC-128 does not possess a universally adopted industrial standard. In this work, CRC-128 is implemented primarily as a generalized software extension intended to evaluate framework flexibility. The CRC state is represented using two 64-bit words:

$$(H, L) \tag{25}$$

corresponding to the high-order and low-order halves of the 128-bit register.

GF(2)-based matrix advancement and combine operations are extended to operate on the full 128-bit state representation. The same polynomial formulation used for smaller CRC families is preserved, while transformation matrices are enlarged accordingly to support extended register width.

5 Experimental Evaluation

This section evaluates the proposed parallel CRC framework in terms of execution time, scalability, throughput, latency, and energy-oriented behavior. The experimental analysis additionally investigates synchronization overhead, memory limitations, and practical multi-core scaling characteristics.

5.1 Experimental Setup

Experiments were conducted on a general-purpose multi-core platform. Table 2 summarizes the experimental environment.

Table 2. Experimental Configuration

Parameter	**Configuration**
Hardware Platform	ASUS U36JC Laptop
CPU	Intel Core i5-2430M
Core Configuration	2 Cores / 4 Threads
Clock Frequency	2.40 GHz
Memory	4 GB RAM
Operating System	Windows 10
Compiler	CompilerGCC (Dev-C++ 6.3), -O2 Optimization
Thread Library	POSIX Threads (pthreads)
CRC Variants	CRC-8,16,32,64,128
Dataset Sizes	100 MB, 500 MB, 1000 MB
Thread Counts	1,2,4,8
Repetitions	10

All measurements were repeated multiple times and averaged to reduce transient operating-system fluctuations.

5.2 Scalability Evaluation

To investigate scalability behavior, experiments were performed under multiple thread configurations. Parallel speedup is defined as:

$$Speedup(T) = \frac{T_{serial}}{T_T} \tag{26}$$

where:

$$T_T \tag{27}$$

denotes execution time obtained using T worker threads. Parallel efficiency is computed as:

$$Efficiency(T) = \frac{Speedup(T)}{T} \tag{28}$$

Table 3 presents scalability results for CRC-32 using a 1000 MB dataset. The reported execution times correspond to the mean values obtained from ten independent experimental runs. Standard deviations were consistently small, indicating that the proposed framework exhibits stable execution behavior with low run-to-run variability under the evaluated experimental conditions.

Table 3. Scalability Results for CRC-32 (Mean ± Standard Deviation, $n = 10$)

Threads	Execution Time (s)	Speedup	Efficiency
1	312.05 ± 1.18	1.00	1.00
2	176.80 ± 0.96	1.76	0.88
4	103.30 ± 0.63	3.02	0.76
8	92.10 ± 0.58	3.39	0.42

The results demonstrate substantial performance improvement when transitioning from serial execution to parallel processing. However, scaling gradually becomes sublinear at higher thread counts because of synchronization cost, cache contention, Hyper-Threading resource sharing, and memory bandwidth limitations.

5.3 Throughput Analysis

Throughput was evaluated using:

$$Throughput = \frac{Processed\ Data}{Execution\ Time} \tag{29}$$

and reported in MB/s.

Table 4 summarizes throughput results. The proposed framework consistently improves throughput for large datasets.

CRC-128 exhibits comparatively lower throughput because larger register width and extended GF(2) transformation operations increase computational cost. The reported throughput values should be interpreted in the context of the experimental platform and implementation objectives. All experiments were conducted on an Intel Core i5-2430M processor using a portable pthread-based software implementation without architecture-specific optimizations such as SIMD vectorization, carry-less multiplication (PCLMULQDQ), or dedicated hardware CRC instructions. In addition, compiler optimizations were limited to the standard GCC optimization level to preserve portability across different platforms. Consequently, the measured throughput is expected to be lower than that reported by highly optimized CRC implementations specifically designed for modern processors with dedicated instruction-set support. The benchmarking methodology was designed to evaluate the relative performance improvement achieved through thread-level parallelization rather than the absolute maximum throughput obtainable on a particular processor architecture. Therefore, identical compiler settings, workload sizes, and execution conditions were maintained for both the serial and parallel implementations to ensure a fair comparison.

Table 4. Throughput Comparison

CRC Type	Serial (MB/s)	Parallel (4 Threads)
CRC-8	3.33	9.60
CRC-16	3.27	9.21
CRC-32	3.20	9.67
CRC-64	2.95	8.84
CRC-128	2.22	5.40

5.4 Comparative Benchmarking Discussion

To better position the proposed framework relative to prior CRC optimization methods, Table 5 summarizes approaches. The acceleration values reported for existing CRC optimization techniques are taken from the corresponding publications and are presented only for qualitative comparison. Since the reported results were obtained using different processor architectures, compiler optimization settings, and benchmarking methodologies, they should not be interpreted as direct experimental comparisons with the proposed framework. The primary objective of this comparison is to position the proposed portable software framework within the broader CRC optimization landscape.

The proposed framework is not intended to outperform specialized hardware-assisted CRC implementations. Instead, its primary objectives are:

- generalized multi-variant support,
- portable software deployment,
- correctness-preserving combine operation,

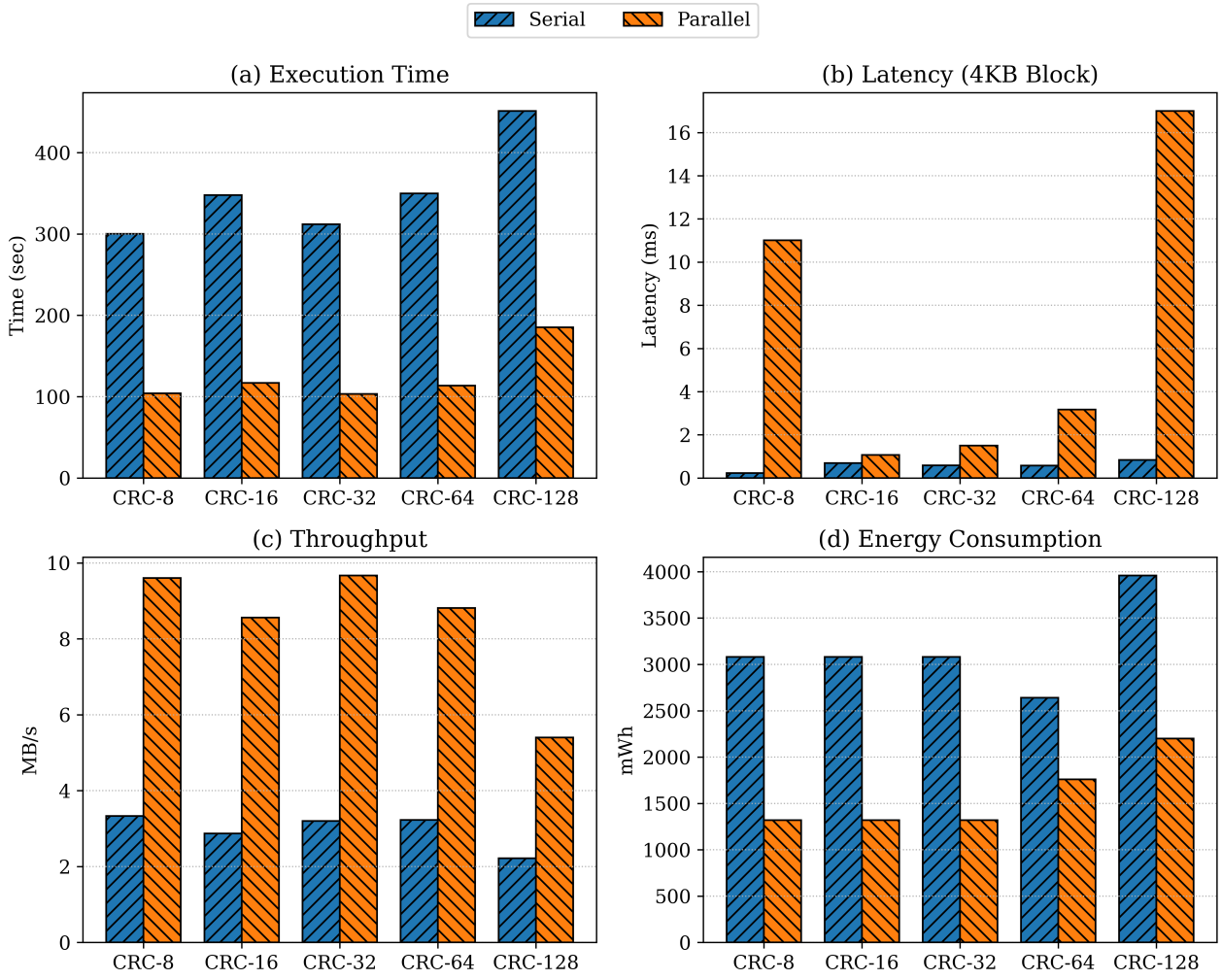


Figure 3. Comparison between serial and parallel CRC in terms of execution time, energy consumption, throughput, and latency for a 1000 MB dataset.

Table 5. Comparison with CRC Optimization Techniques Reported in the Literature

Method	**Reported Acceleration**	**Reference**	**Remarks**
Serial CRC	1×	–	Baseline
Lookup-Table CRC	2–5×	[7]	Portable software
Slicing-by-8	5–20×	[7], [15]	Mainly CRC-32
SIMD / Vectorized CRC	10–40×	[6], [15]	Processor-specific (SSE/AVX)
Hardware CRC Instructions	20–50×	[6]	Dedicated CRC instructions
Proposed Framework	3–4×	**This work**	**Portable multi-CRC framework**

- software-oriented multi-core evaluation.

5.5 Latency Evaluation

Latency behavior was examined using small message blocks. Latency was computed using a 4 KB workload and averaged over ten independent executions. Table 6 reports latency results.

Although throughput improves significantly for large datasets, parallel execution introduces additional thread creation, scheduling, and synchronization overhead. Consequently, serial CRC execution may remain preferable for:

Table 6. Latency Results

CRC Type	Serial (ms)	Parallel (ms)
CRC-8	0.228	11.00
CRC-16	0.244	10.92
CRC-32	0.255	9.85
CRC-64	0.317	11.42
CRC-128	0.551	14.87

- small payload sizes,
- ultra-low-latency applications,
- embedded control systems.

5.6 Energy Evaluation

Energy analysis was performed using battery discharge monitoring under controlled experimental conditions. Estimated energy consumption was approximated using:

$$Energy(J) \approx Capacity(mWh) \times \Delta Battery(\%) \times 3.6 \quad (30)$$

The approximation symbol is intentionally used because battery-percentage monitoring does not provide direct processor-level power instrumentation. Table 7 summarizes energy observations. It should be noted that the energy values reported in Table 7 correspond to the estimated total energy consumed for processing the complete dataset used in each experiment rather than a single CRC operation. Therefore, the reported values should be interpreted as workload-level energy measurements. Since CRC computations were performed repeatedly over large datasets (100 MB, 500 MB, and 1000 MB), the measured energy reflects the cumulative cost of millions of CRC processing steps. Consequently, these results are intended for comparative evaluation between serial and parallel implementations rather than for estimating the energy of an individual CRC calculation.

Table 7. Energy Comparison

CRC Type	Serial	Parallel
CRC-8	2890	1510
CRC-16	2960	1450
CRC-32	3080	1320
CRC-64	3370	1690
CRC-128	4210	2210

Despite higher instantaneous CPU utilization, shorter execution time leads to lower estimated total energy usage for large workloads. However, these results should be interpreted as comparative estimates rather than precise physical power measurements. Future work will employ processor-level instrumentation tools such as Intel RAPL, performance counters, and dedicated power monitors.

5.7 Scalability Discussion and Amdahl Analysis

Observed scaling behavior is additionally examined through Amdahl's Law. Assuming a parallel fraction:

$$P = 0.92 \quad (31)$$

Theoretical speedup becomes:

$$S(T) = \frac{1}{(1-P) + \frac{P}{T}} \quad (32)$$

Table 8 compares theoretical and measured speedup behavior.

Table 8. Amdahl Comparison

Threads	Theoretical	Measured
1	1.00	1.00
2	1.85	1.76
4	3.13	3.02
8	4.71	3.39

Deviation from ideal scaling is attributed primarily to:

- synchronization overhead,
- cache effects,
- shared execution resources,
- memory bandwidth saturation.

The experimental results are summarized in Fig. 3 a-d, which presents a comprehensive comparison between serial and parallel implementations across all CRC types.

5.8 Limitations

Several limitations should be acknowledged. First, experiments were conducted on a legacy dual-core platform. Second, battery-based energy estimation provides approximate comparative information rather than high-precision power measurement. Third, comparison against highly optimized instruction-level implementations remains limited. Finally, CRC-128 is included primarily as a generalized software extension rather than a standardized industrial CRC specification. These limitations motivate future work involving modern many-core architectures, hardware-assisted

CRC benchmarking, and processor-level energy instrumentation. Although the scalability analysis presented in this work is supported by Amdahl's Law and experimental measurements on the evaluated dual-core platform, scalability beyond the tested hardware remains analytical rather than experimentally validated. Evaluating the proposed framework on modern multi-core and many-core processors would enable a more comprehensive investigation of memory bandwidth saturation, cache contention, synchronization overhead, and scalability limits under substantially higher core counts. Such an experimental study constitutes an important direction for future work.

5.9 Impact of Dataset Size

To investigate the effect of workload size on the effectiveness of parallel CRC computation, experiments were conducted using three dataset sizes: 100 MB, 500 MB, and 1000 MB. The obtained results indicate that the benefits of parallelization become increasingly significant as the input size grows.

For the 100 MB dataset, the proposed framework already provides noticeable performance improvement compared with serial execution. However, the relative gain remains limited because thread creation, synchronization, and CRC combination overhead represent a larger fraction of the total execution time. Consequently, parallel efficiency is reduced for smaller workloads.

For the 500 MB dataset, computation time becomes the dominant component of execution cost, and the overhead associated with thread management is amortized over a larger amount of processed data. As a result, higher speedup and throughput are achieved, while estimated energy consumption is reduced because the overall execution time decreases.

The largest dataset (1000 MB) exhibits the highest benefit from parallel execution. In this case, thread-management overhead becomes comparatively negligible relative to the total computation workload. The framework achieves its best throughput and speedup values while maintaining exact CRC correctness. The energy measurements also indicate greater overall efficiency due to the shorter processing time.

These observations demonstrate that the proposed framework is most advantageous for large-scale data processing applications, where the computational workload substantially exceeds synchronization and scheduling overhead. For smaller datasets, the performance gains remain positive but are partially limited by thread-management costs. Table 9 shows the impact of workload size on the effectiveness of the proposed parallel CRC framework. As the dataset size increases, the relative contribution of thread creation, synchronization, and CRC combination overhead decreases compared with the total computation time. Consequently, larger datasets benefit more from parallel execution and achieve higher speedup values. The results indicate that the proposed framework is particularly suitable for large-scale data processing applications, where computational workload dominates synchronization overhead.

Table 9. Effect of Dataset Size on Parallel CRC Performance

Dataset Size	Serial Time (s)	Parallel Time (s)	Speedup
100 MB	31.20	11.00	2.84
500 MB	156.70	55.20	2.84
1000 MB	312.05	103.30	3.02

6 Conclusions

This paper presented a generalized software-based framework for parallel CRC computation on multi-core processors using POSIX threads (pthreads), supporting multiple CRC variants including CRC-8, CRC-16, CRC-32, CRC-64, and CRC-128 within a unified implementation model. To ensure correctness, a GF(2)-based combination mechanism derived from polynomial arithmetic and matrix-based shifting operations was employed, guaranteeing equivalence between serial and parallel CRC computation. Experimental results demonstrated that the proposed framework significantly improves execution time and throughput for large datasets while preserving exact CRC correctness, achieving notable speedup on the evaluated platform. The analysis further highlighted the trade-off between throughput and latency, showing that although parallel execution is beneficial for large workloads, synchronization and thread-management overhead can limit efficiency for smaller inputs. Scalability behavior was found to be consistent with Amdahl's Law and influenced by practical factors such as synchronization cost, cache contention, memory bandwidth limitations, and shared hardware resources. In addition, energy-oriented evaluation indicated that reduced execution time can lower overall energy consumption despite increased instantaneous processor utilization. While the proposed framework is not intended to compete directly with highly optimized hardware-assisted, SIMD-based, or instruction-level CRC acceleration techniques, it provides a portable, correctness-preserving, and extensible software solution for parallel CRC processing across multiple CRC families. Future research will focus on evalua-

tion over modern many-core platforms, comparison with advanced CRC optimization techniques, adaptive workload distribution strategies, and more accurate processor-level energy measurement methodologies.

Future research will focus on extensive experimental evaluation over modern multi-core and many-core platforms to validate the scalability of the proposed framework under higher core counts and to further investigate the impact of memory bandwidth, cache contention, and synchronization overhead on large-scale parallel CRC computation.

Table 10. Notation Used in the Proposed Framework

Symbol	Description
D	Input dataset
L	Total dataset size in bytes
T	Number of worker threads
D_i	Data chunk assigned to thread i
$Start_t$	Starting index of thread t
End_t	Ending index of thread t
$P(x)$	CRC generator polynomial
c_1	CRC of first chunk
c_2	CRC of second chunk
l_2	Length of second chunk
M	GF(2) transition matrix
$CRC_{partial}$	Partial CRC result computed by a thread
CRC_{final}	Final combined CRC value

References

[1] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, jan 1961. doi:10.1109/JRPROC.1961.287814.

[2] Charanarur Panem, Vinaya Gad, and Rajendra S. Gad. Polynomials in error detection and correction in data communication system. In *Coding Theory*, pages 5–6. IntechOpen, 2019. doi:10.5772/intechopen.86160.

[3] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, 9 edition, 2025.

[4] Ming-Der ShiehMing-Hwa SheuChung-Ho ChenHsin-Fu Lo. A systematic approach for parallel crc computations. *Journal of Information Science and Engineering*, 17(3):445–461, 2001. doi:10.6688/JISE.2001.17.3.3.

[5] M. Russo G. Campobello, G. Patane. Parallel crc realization. *IEEE Transactions on Computers*, 52(10):1312–1319, oct 2003. doi:10.1109/TC.2003.1234528.

[6] Vinodh Gopal, Erdinc Ozturk, Jim Guilford, Gil Wolrich, Wajdi Feghali, and Martin Dixon. Fast crc computation for generic polynomials using pclmulqdq instruction, 2009. URL https://api.semanticscholar.org/CorpusID:6753239.

[7] Andrew Kadatch1 and Bob Jenkins. Everything we know about crc but afraid to forget, 2010. URL https://github.com/tpn/pdfs/blob/master/Everything%20We%20Know%20About%20CRC%20But%20Afraid%20To%20Forget%20(3rd%20September%2C%202010).pdf.

[8] Sam Russell. Chorba: A novel crc32 implementation. arXiv preprint, 2024.

[9] C. Chen, R. Tian, and Z. Yao. Design and implementation of the crc-32 checksum of parallel algorithm. *Chinese Journal of Scientific Instrument*, 34:151–155, 2013. doi:10.1109/54.922808.

[10] Adler M. "zlib compression library, 2026. URL https://zlib.net.

[11] Linux Kernel Documentation. Crc32 and crc32c implementations, 2026. URL https://www.kernel.org.

[12] Aiwei Lei and Ye Yu. Development optimization and future direction of crc checking technology. In *Proceedings of the 3rd International Conference on Mechanical Engineering and Electrical Automation (ICMEEA 2025)*, pages 1–6, 2025. doi:10.2991/978-94-6463-821-9_76.

[13] G. Albertengo and R. Sisto. Parallel crc generation. *IEEE Micro*, 10(5):63–71, oct 1990. doi:10.1109/40.60527.

[14] Mathys Walma. Pipelined feed-forward cyclic redundancy check (crc) calculation. arXiv preprint, 2006.

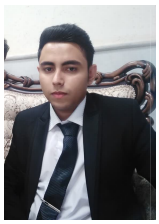
[15] Nicolas Gorius andGeorge Nehmetallah Dat Tran, Shahid Aslam. Parallel computation of crc-code on an fpga platform for high data throughput. *Electronics*, 10(4):439–445, 2021. doi:10.3390/electronics10070866.

[16] Ling Zhang, Shanwei Ye, Zhuo Gou, Xuefei Yang, Qilin Dai, Fuqiang Wang, and Yingcheng Lin. An efficient parallel crc computing method for high bandwidth networks and fpga implementation. *Electronics*, 13:4399, 2024. doi:10.3390/electronics13224399. Article 4399, pp. 1–10.

[17] T.-B. Pei and C. Zukowski. High-speed parallel crc circuits in vlsi. *IEEE Transactions on Communications*, 40(4):653–657, apr 1992. doi:10.1109/26.141415.

[18] C. Cheng and K. K. Parhi. High-speed parallel crc implementation based on unfolding, pipelining, and retiming. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 53(10):1017–1021, oct 2006. doi:10.1109/TCSII.2006.882213.

[19] Jiaqiang Li, Shanshan Liu, Pedro Reviriego, Liyi Xiao, and Fabrizio Lombardi. Efficient concurrent fault detection for parallel cyclic redundancy check circuits. *IET Computers & Digital Techniques*, 14:678–685, 2020. doi:10.1049/iet-cdt.2018.5183.



Mohammadjavad Khani received the B.Sc. and M.Sc. degrees in Computer Engineering from Shahid Rajaee Teacher Training University, Tehran, Iran, in 2018 and 2022, respectively. He is currently pursuing the Ph.D. degree in Computer Engineering at Razi University, Kermanshah, Iran. He is also a Visiting Lecturer at Razi University and a Technical Teacher with the Ministry of Education, Iran. His research interests include artificial intelligence, machine learning, parallel and high-performance computing, GPU programming, wireless sensor networks, and energy-efficient computing. He has authored several journal and conference papers in these research areas.

Mahmood Ahmadi received the B.S. degree in Computer Engineering from the University of Isfahan, Isfahan, Iran, in 1995. He received the M.Sc. degree in Computer Architecture and Engineering from Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, in 1998. From 1999 to 2005, he was a faculty member at Razi University in Kermanshah. In October 2005, he joined the Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS) at Delft University of Technology, Delft, The Netherlands, as a full-time Ph.D. student. He received his Ph.D. in May 2010. His research interests include Software-Defined Networking, Named-Data Networking, Network Function Virtualization, Fog and Cloud Computing, the Internet of Things, Intrusion Detection, and High-Performance Computing. He has published more than 110 journal and conference papers. He is currently a Professor at the Computer Engineering Department, Razi University, Kermanshah, Iran.