CSE 4001

Parallel and Distributed Computing

Project Report

Parallelization of Cryptocurrency Block-Mining Using OpenMP

Winter Semester 2022-23

Submitted By : Anish Desai 20BCE0461
Akshat Bisht 20BCE0629
Kartikeya Rawat 20BCE0641
Rajat Lohia 20BCE2298

Under the guidance of : Prof. Balamurugan R.

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Table of Contents

I. Abstract…………………………………………………………………...3
II. List of Symbols, Abbreviations or Nomenclature………………………..3
III. Chapters
1. Introduction…………………………………...………………...4
Tech Stack Used………………………………………………...4
2. Literature Review...……………………………………………..5
3. Problem Formulation…………………………………………..10
4. Implementation
i. SHA256 ………………………………………………...11
ii. SHA256 Serial Implementation………………………...11
iii. SHA256 Parallel Implementation………………………14
iv. SHA256 Algorithm & Explanation………………………18
v. SHA256 Diagrammatic Flowchart………………………..19
vi. Blockchain Mining……………………………………...20
vii. Blockchain Mining Serial Implementation………………..20
viii. Blockchain Mining Parallel Implementation……………...22
ix. Blockchain Mining Algorithm & Explanation……………23
x. Blockchain Mining Flowchart……………………………24
IV. Results
1. Implementation Output………………………………………..25
2. Comparative Statistical Analysis……………………………...27
V. Conclusions and Future work..…………………………………………34
VI. Appendices/Annexures…………………………………………………35
VII. References………………………………………………………………37

ABSTRACT

In this age of new technologies, blockchain is bringing significant disruption in a spectrum of fields ranging from banking to education, healthcare to even governance. This is possible through cryptographically-secure 'hashing'. Miners compete to mine out blocks in the least time possible, but have proportionally complex computations to perform such as finding the nonce and hash values and the application of hashing algorithm, especially SHA-256 algorithm twice during the process. The computations can be greatly simplified if performed concurrently and help blockchain as a concept make great strides as a potential technology. In our project, we propose to implement the parallelization of cryptocurrency block mining, i.e., hashing function of a cryptocurrency and deliver a discrete version of the model. For this, we propose to use parallelization and cryptographic tools, and adopt a brute-force method.

LIST OF SYMBOLS, ABBREVIATIONS OR NOMENCLATURE

SHA256 *– Secure Hash Algorithm with a fixed-length output of 256 bits*
Txn. *– Transaction*
OpenMP *– Open Multi-processing, an API for developing parallel programs*
PoW *– 'Proof of Work', a consensus mechanism in bitcoin network*

INTRODUCTION

Blockchain technology is a distributed ledger system that enables secure and decentralized transactions without the need for intermediaries. The security of the blockchain network relies on the proof-of-work consensus algorithm, which involves miners solving complex mathematical problems to validate transactions and add new blocks to the chain. However, the computational power required for mining blocks has significantly increased over time, making it a resource-intensive process. To improve the efficiency of the mining process, parallelization techniques have been employed to enable multiple miners to work on different parts of the blockchain simultaneously. To tackle this issue, parallelization techniques have been employed to enhance the efficiency of the mining process. In this research paper, we present the parallelization of blockchain mining using OpenMP, a widely-used shared-memory parallel programming model. We have used CodeBlocks IDE for the implementation, and SHA256 has been parallelized to improve the hashing process. Additionally, the overall mining process has also been parallelized to reduce the time required for block creation. Our experimental results demonstrate that parallelization of blockchain mining using OpenMP can significantly improve the efficiency of the mining process, thereby enhancing the scalability and security of the blockchain network.

Tech Stack Used

- CodeBlocks IDE
- OpenMP for Parallelization
- Standard OpenMP Header Files

LITERATURE REVIEW

Sno.	Research Paper	Summary	Findings/Result	Limitations
1.	"A Parallel Proof of Work to Improve Transaction Speed and Scalability in Blockchain Systems". Shihab Shahriar Hazari, Qusay H. Mahmoud. IEEE 2019. [1]	The proposed method includes a process for selection of a manager, distribution of work and a reward system. This method has been implemented in a test environment that contains all the characteristics needed to perform Proof of Work for Bitcoin and has been tested, using a variety of case scenarios, by varying the difficulty level and number of validators.	Preliminary results show improvement in the scalability of Proof of Work up to 34% compared to the current system.	The paper only focuses on improving the PoW consensus algorithm in blockchain systems. Other consensus algorithms haven't been considered. The paper does not provide a detailed evaluation of the proposed algorithm's security implications. The paper does not provide a comparison of the proposed algorithm's performance with existing parallelization techniques in blockchain systems.
2.	"DiPETrans: A Framework	The objective is to increase the transaction throughput by	The mean speedup increases as the number of transactions per	Serial outperforms 1-parallel configuration due

	for Distributed Parallel Execution of Transactions of Blocks in Blockchain”. Shrey Baheti, Parwat Singh Anjana, Sathya Peri and Yogesh Simmhan. Wiley 2021. [2]	introducing parallel transaction execution using a static analysis over the transaction dependencies. A DiPETrans framework has been proposed for distributed execution of transactions in a block. Peers in the blockchain network form a community of trusted nodes to execute the transactions and find the PoW in-parallel, using a leader–follower approach. During mining, the leader statically analyses the transactions, creates different groups (shards) of independent transactions, and distributes them to followers to execute concurrently. After execution, the community’s compute power is utilized to solve the PoW concurrently.	block increases. Speedup increases till 1/8 and then decreases if there is further decrease in the number of contract transactions per block.	to static analysis and communication overhead
3.	“Parallel mining in blockchain for	Uses game theory to generate nonce value in efficient and	Parallel mining helps to increase the hash rate to mines new and	Lacks details about the implementation

	Bitcoin using game theory". Milind Tote, et. Al. JETIR 2019. [3]	faster way using multiple threads with the help of parallel mining.	valid blocks for the Bitcoin. Several threads are engaged in a competitive way so that a low hash function can be achieved if and only if the generated hash function is unique and not repeated in the decentralized list.	and subsequent results.
4.	"ParBlockch-ain: Leveraging Transaction Parallelism in Permissione-d Blockchain Systems". Mohammad Javad Amiri, Divyakant Agrawal, Amr El Abbadi. DeepAI 2019. [4]	OXII, a new paradigm for permissioned blockchains to support distributed applications that execute concurrently has been introduced. OXII is designed for workloads with (different degrees of) contention. We then present ParBlockchain, a permissioned blockchain designed specifically in the OXII paradigm.	The evaluation of ParBlockchain using a series of benchmarks reveals that its performance in workloads with any degree of contention is better than the state of the art permissioned blockchain systems. 70% more throughput and 7.5 times less latency in comparison to other blockchain systems.	Scalability issues Fault Tolerance issues

5.	“A High-Performance Parallel Hardware Architecture of SHA-256 Hash in ASIC”. Ruizhen Wu; Xiaoyong Zhang; Mingming Wang; Lin Wang. IEEE 2020. [5]	The research paper proposes a high-performance parallel hardware architecture for the SHA-256 hash function, implemented in an ASIC design. The architecture is optimized for high throughput and low latency, with a fully pipelined design and parallel processing of message blocks. The proposed design is capable of achieving a throughput of 7.86 Gbps and a latency of 0.59 µs for processing a 512-bit message block. The design is also scalable, allowing for the parallel processing of multiple message blocks simultaneously.	The proposed design shows significant improvements in terms of throughput and latency compared to these existing implementations, making it well-suited for high-performance applications such as network security and digital currency mining.	Lack of a detailed analysis of the power consumption and area requirements of the proposed design, which are important factors for practical ASIC implementations. Additionally, the authors do not provide a comprehensive analysis of the security properties of the proposed design, although they do note that it is compliant with the standard SHA-256 algorithm.
6.	“Efficient parallel execution of block transactions in blockchain”. Parwat Singh Anjana.	The research paper proposes an efficient parallel execution mechanism for block transactions in a blockchain. The proposed mechanism uses a novel approach that parallelizes the	The paper presents results of experiments conducted on a simulated blockchain network, demonstrating the improved performance and	Experiments were conducted on a simulated blockchain network, and therefore, the results may not be representative of real-world scenarios.

	ACM 2021. [6]	validation of transactions within a block, resulting in faster processing times and improved scalability. The mechanism also includes a priority queue that orders transactions based on their priority, ensuring that higher priority transactions are validated first.	scalability of the proposed mechanism compared to existing approaches. The experiments show that the proposed mechanism can achieve up to 20% faster block processing times and a significant reduction in the number of unprocessed transactions compared to existing approaches.	Additionally, the paper does not provide a detailed analysis of the security implications of the proposed mechanism or its potential impact on the overall security of the blockchain network.
7.	"Parallel SHA-256 on SW26010 many-core processor for hashing of multiple messages". Ziheng Wang, Xiaoshe Dong, Yan Kang & Heng Chen. Springer 2023. [7]	The research paper presents an implementation of the SHA-256 hash function using parallel processing on a SW26010 many-core processor. The proposed implementation is optimized for hashing multiple messages simultaneously, with each message processed independently and in parallel.	The paper presents experimental results comparing the proposed implementation with existing software implementations of the SHA-256 hash function on a range of hardware platforms. The results demonstrate that the proposed implementation outperforms	The limitations of the paper include the fact that the experiments were conducted on a specific hardware platform, and the results may not be directly applicable to other hardware configurations. Additionally, the paper does not provide a comprehensive analysis of the security properties of the

			existing implementations, achieving higher throughput and lower processing times.	proposed implementation, although the authors note that it is compliant with the standard SHA-256 algorithm.

Table 3.2 : Tabular Literature Survey

PROBLEM FORMULATION

The Blockchain technology has gained immense popularity in recent years due to its potential for providing a decentralized and secure platform for various applications, including cryptocurrency transactions. However, the traditional PoW algorithm used in the Blockchain mining process requires significant computational resources, making it a time-consuming and energy-intensive task. The SHA256 algorithm, which is used for hashing the transaction data in the mining process, is also computationally intensive. As the size of the Blockchain network continues to grow, the scalability and efficiency of the mining process become increasingly important. Therefore, the problem addressed by this research project is the need to improve the efficiency and scalability dimensions of the Blockchain mining process and the SHA256 algorithm by implementing parallelization techniques using OpenMP. This can be achieved by addressing the issue of latency in execution of the algorithm and mining processes.

IMPLEMENTATION

The **SHA-256 algorithm** is a cryptographic hash function that is used in various applications, including digital signatures, data integrity checks, and password hashing. It is a one-way function that takes an input and produces a fixed-size, unique output called a hash.

In the context of blockchain mining, SHA-256 plays a crucial role in the process of verifying and adding new transactions to the blockchain. To mine a new block, miners compete to solve a complex mathematical puzzle based on the SHA-256 algorithm. The first miner to solve the puzzle and validate the transactions in the block is rewarded with newly minted cryptocurrency.

In particular, the mining process involves repeatedly applying the SHA-256 algorithm to a block of transaction data, along with a random value known as a "nonce," until a hash with a certain number of leading zeros is obtained. This process requires significant computational power and energy consumption, as the algorithm is designed to be difficult to solve but easy to verify.

SHA256 Serial Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <time.h>

#define ROTR(x, n) (((x) >> (n)) | ((x) << (32 - (n))))
#define SHR(x, n) ((x) >> (n))
#define CH(x, y, z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define SIGMA0(x) (ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define SIGMA1(x) (ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25))
#define DELTA0(x) (ROTR(x, 7) ^ ROTR(x, 18) ^ SHR(x, 3))
#define DELTA1(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ SHR(x, 10))

static const uint32_t K[] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
```

```c
    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};

void sha256_transform(uint32_t *state, const uint32_t *block)
{
    uint32_t W[64];
    uint32_t a, b, c, d, e, f, g, h, T1, T2;
    int i;

    for (i = 0; i < 16; i++)
    {
        W[i] = block[i];
    }
    for (i = 16; i < 64; i++)
    {
        W[i] = DELTA1(W[i - 2]) + W[i - 7] + DELTA0(W[i - 15]) + W[i - 16];
    }

    a = state[0];
    b = state[1];
    c = state[2];
    d = state[3];
    e = state[4];
    f = state[5];
    g = state[6];
    h = state[7];
    for (i = 0; i < 64; i++)
    {
        T1 = h + SIGMA1(e) + CH(e, f, g) + K[i] + W[i];
        T2 = SIGMA0(a) + MAJ(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;
    }

    state[0] += a;
```

```
    state[1] += b;
    state[2] += c;
    state[3] += d;
    state[4] += e;
    state[5] += f;
    state[6] += g;
    state[7] += h;
}

void sha256_hash(const unsigned char *message, size_t len, unsigned char
*digest)
{
    uint32_t state[8] = {
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19};
    uint32_t block[16];
    size_t i, j;

    for (i = 0; i + 64 <= len; i += 64)
    {
        for (j = 0; j < 16; j++)
        {
            block[j] = (uint32_t)(message[i + j * 4 + 0]) << 24 |
                       (uint32_t)(message[i + j * 4 + 1]) << 16 |
                       (uint32_t)(message[i + j * 4 + 2]) << 8 |
                       (uint32_t)(message[i + j * 4 + 3]) << 0;
        }
        sha256_transform(state, block);
    }
    memset(block, 0, sizeof(block));
    for (j = 0; i + j < len; j++)
    {
        block[j / 4] |= (uint32_t)(message[i + j]) << (24 - j % 4 * 8);
    }
    block[j / 4] |= (uint32_t)0x80 << (24 - j % 4 * 8);
    if (j >= 56)
    {
        sha256_transform(state, block);
        memset(block, 0, sizeof(block));
    }
    block[15] = (uint32_t)len << 3;
    sha256_transform(state, block);

    for (i = 0; i < 8; i++)
    {
        digest[i * 4 + 0] = (unsigned char)(state[i] >> 24);
        digest[i * 4 + 1] = (unsigned char)(state[i] >> 16);
        digest[i * 4 + 2] = (unsigned char)(state[i] >> 8);
```

```c
        digest[i * 4 + 3] = (unsigned char)(state[i] >> 0);
    }
}
int main()
{
    const char *message = "Okay";
    unsigned char digest[32];

    clock_t start_time = clock();

    sha256_hash((unsigned char *)message, strlen(message), digest);

    int i;
    printf("The plaintext is: %s",message);
    printf("\n");
    printf("The hashed digest is: ");
    for (i = 0; i < 32; i++)
    {
        printf("%02x", digest[i]);
    }
    printf("\n");

    clock_t end_time = clock();
    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", elapsed_time);

    return 0;
}
```

SHA256 Parallel Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include <stdint.h>

#define ROTR(x, n) (((x) >> (n)) | ((x) << (32 - (n))))
#define SHR(x, n) ((x) >> (n))
#define CH(x, y, z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define SIGMA0(x) (ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define SIGMA1(x) (ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25))
```

```c
#define DELTA0(x) (ROTR(x, 7) ^ ROTR(x, 18) ^ SHR(x, 3))
#define DELTA1(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ SHR(x, 10))

static const uint32_t K[] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};

void sha256_transform(uint32_t *state, const uint32_t *block)
{
    uint32_t W[64];
    uint32_t a, b, c, d, e, f, g, h, T1, T2;
    int i;

    for (i = 0; i < 16; i++)
    {
        W[i] = block[i];
    }
    for (i = 16; i < 64; i++)
    {
        W[i] = DELTA1(W[i - 2]) + W[i - 7] + DELTA0(W[i - 15]) + W[i - 16];
    }

    a = state[0];
    b = state[1];
    c = state[2];
    d = state[3];
    e = state[4];
    f = state[5];
    g = state[6];
    h = state[7];

#pragma omp parallel for private(T1, T2)
    for (i = 0; i < 64; i++)
    {
```

```c
        T1 = h + SIGMA1(e) + CH(e, f, g) + K[i] + W[i];
        T2 = SIGMA0(a) + MAJ(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;
    }

    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
    state[4] += e;
    state[5] += f;
    state[6] += g;
    state[7] += h;
}

void sha256_hash(const unsigned char *message, size_t len, unsigned char
*digest)
{
    uint32_t state[8] = {
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19};
    uint32_t block[16];
    size_t i, j;

    for (i = 0; i + 64 <= len; i += 64)
    {
        for (j = 0; j < 16; j++)
        {
            block[j] = (uint32_t)(message[i + j * 4 + 0]) << 24 |
                       (uint32_t)(message[i + j * 4 + 1]) << 16 |
                       (uint32_t)(message[i + j * 4 + 2]) << 8 |
                       (uint32_t)(message[i + j * 4 + 3]) << 0;
        }
        sha256_transform(state, block);
    }
    memset(block, 0, sizeof(block));
    for (j = 0; i + j < len; j++)
    {
        block[j / 4] |= (uint32_t)(message[i + j]) << (24 - j % 4 * 8);
    }
    block[j / 4] |= (uint32_t)0x80 << (24 - j % 4 * 8);
```

```c
    if (j >= 56)
    {
        sha256_transform(state, block);
        memset(block, 0, sizeof(block));
    }
    block[15] = (uint32_t)len << 3;
    sha256_transform(state, block);

    for (i = 0; i < 8; i++)
    {
        digest[i * 4 + 0] = (unsigned char)(state[i] >> 24);
        digest[i * 4 + 1] = (unsigned char)(state[i] >> 16);
        digest[i * 4 + 2] = (unsigned char)(state[i] >> 8);
        digest[i * 4 + 3] = (unsigned char)(state[i] >> 0);
    }
}
int main()
{
    const char *message = "Okay";
    unsigned char digest[32];
    double start_time, end_time;

    start_time = omp_get_wtime();
    sha256_hash((unsigned char *)message, strlen(message), digest);

    int i;
    printf("The plaintext is: %s",message);
    printf("\n");
    printf("The hashed digest is: ");
    for (i = 0; i < 32; i++)
    {
        printf("%02x", digest[i]);
    }
    printf("\n");

    end_time = omp_get_wtime();

    printf("Time taken: %f seconds\n", end_time - start_time);

    return 0;
}
```

Algorithm Explanation

The code includes necessary header files such as <stdio.h>, <stdlib.h>, <string.h>, <omp.h>, and <stdint.h>. These headers provide useful functions and data types that will be used in the program.

Some macros are defined at the top of the code. These macros are used to define certain operations such as right rotation, shifting, and logical operations on the bits of the input values.

A constant array K is declared and initialized with predefined values. These values will be used in the SHA-256 algorithm.

The sha256_transform function is defined. This function takes two arguments, state and block, both of which are arrays of type uint32_t.

The function first declares an array of 64 uint32_t elements called W and initializes the first 16 elements of W with the values from the block array.

The function then uses a loop to compute the remaining 48 elements of W using the DELTA0, DELTA1, and right rotation macros.

The function then declares 8 uint32_t variables a, b, c, d, e, f, g, and h, and initializes them to the values in the state array.

The function then enters an OpenMP parallel for loop that iterates 64 times. Each iteration of the loop computes the values of T1 and T2 using the SIGMA0, SIGMA1, CH, MAJ, and K macros.

The loop then updates the values of a, b, c, d, e, f, g, and h according to the SHA-256 algorithm.

After the loop completes, the function updates the values in the state array with the updated values of a, b, c, d, e, f, g, and h.

The main function is defined, which takes no arguments and returns an int.

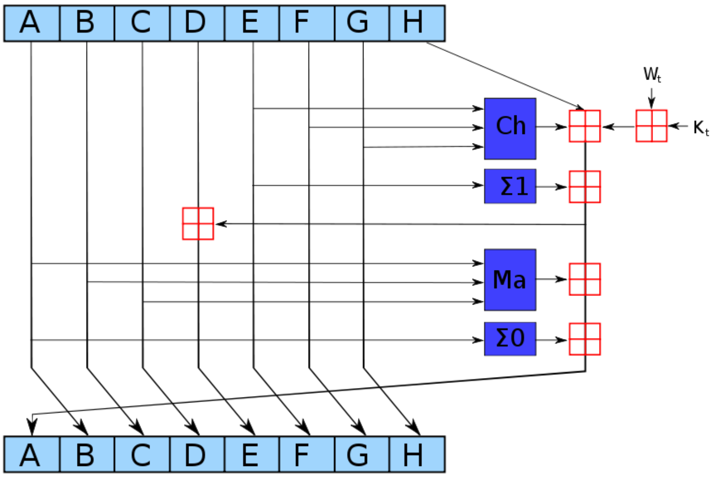
The main function declares an array called message of type char, which will hold the input message to be hashed.

The function then prompts the user to enter a message to be hashed.

The function then declares an array called hash of type uint32_t, which will hold the resulting hash value.

The function then calls the sha256_transform function to compute the hash value of the input message.

Finally, the main function prints out the resulting hash value in hexadecimal format.

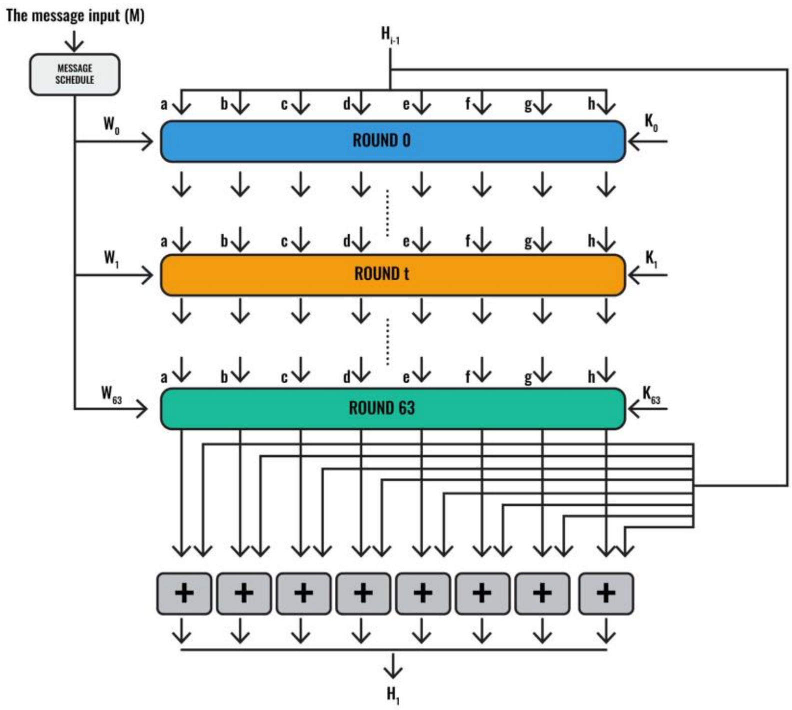


The values of a,b,c,d,e,f,g, and h loop over each round of the SHA256 algorithm.

This represents the computations performed on the buffers in each round.

Fig. 3.4.5.1 : Computations of each round of SHA256

Fig. 3.4.5.2 : Overall working of SHA256 Algorithm



Blockchain mining is the process by which new transactions are verified and added to the public ledger of a blockchain network. It involves using specialized computer hardware and software to solve complex mathematical puzzles and verify transactions on the network.

When a new transaction is made on a blockchain network, it is added to a pool of unconfirmed transactions. Miners then compete to verify these transactions and add them to the blockchain by solving a mathematical puzzle. The first miner to solve the puzzle and verify the transaction is rewarded with newly minted cryptocurrency or transaction fees, depending on the specific blockchain network.

Blockchain Mining Serial Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>

// Define the difficulty level of the mining algorithm
#define DIFFICULTY_LEVEL 4

// Define the maximum nonce value
#define MAX_NONCE 101

// Define the block structure
typedef struct
{
    uint32_t index;
    uint32_t timestamp;
    char data[256];
    uint32_t previous_hash;
    uint32_t hash;
} Block;

// Define a function to calculate the hash of a block
uint32_t calculate_hash(Block *block)
{
    char block_data[1024];
    sprintf(block_data, "%d%d%s%d", block->index, block->timestamp, block-
>data, block->previous_hash);

    uint32_t hash = 0;
    for (int i = 0; i < strlen(block_data); i++)
    {
        hash += block_data[i];
        hash += (hash << 10);
```

```c
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}

int main()
{
    // Create a new block
    Block block;
    block.index = 1;
    block.timestamp = 123456789;
    strcpy(block.data, "Hello, world!");
    block.previous_hash = 0;

    // Set up the mining loop
    uint32_t nonce = 0;
    uint32_t hash = 0;
    int mining_complete = 0;

    clock_t start_time = clock();

    for (nonce = 0; nonce < MAX_NONCE; nonce++)
    {
        if (nonce == mining_complete)
        {
            continue;
        }
        block.hash = calculate_hash(&block);
        if (block.hash % (1 << (32 - DIFFICULTY_LEVEL)) != 0)
        {
            {
                printf("Block mined: %d\n", nonce);
                mining_complete = 1;
            }
        }
    }

    clock_t end_time = clock();
    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Block Mining time taken: %f seconds\n", elapsed_time);

    return 0;
```

```
}
```

Blockchain Mining Parallel Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <omp.h>

// Define the difficulty level of the mining algorithm
#define DIFFICULTY_LEVEL 4

// Define the maximum nonce value
#define MAX_NONCE 101

// Define the block structure
typedef struct
{
    uint32_t index;
    uint32_t timestamp;
    char data[256];
    uint32_t previous_hash;
    uint32_t hash;
} Block;

// Define a function to calculate the hash of a block
uint32_t calculate_hash(Block *block)
{
    char block_data[1024];
    sprintf(block_data, "%d%d%s%d", block->index, block->timestamp, block-
>data, block->previous_hash);

    uint32_t hash = 0;
    for (int i = 0; i < strlen(block_data); i++)
    {
        hash += block_data[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}
```

```
int main()
{
    // Create a new block
    Block block;
    block.index = 1;
    block.timestamp = 123456789;
    strcpy(block.data, "Hello, world!");
    block.previous_hash = 0;

    // Set up the mining loop
    uint32_t nonce = 0;
    uint32_t hash = 0;
    int mining_complete = 0;

    double start_time = omp_get_wtime();
#pragma omp parallel for private(nonce, hash) shared(mining_complete)
    for (nonce = 0; nonce < MAX_NONCE; nonce++)
    {
        if (nonce == mining_complete)
        {
            continue;
        }
        block.hash = calculate_hash(&block);
        if (block.hash % (1 << (32 - DIFFICULTY_LEVEL)) != 0)
        {
#pragma omp critical
            {
                printf("Block mined: %d\n", nonce);
                mining_complete = 1;
            }
        }
    }

    double end_time = omp_get_wtime();
    printf("Block Mining time taken: %f seconds\n", end_time - start_time);

    return 0;
}
```

Algorithm & Explanation

The program defines a block structure, which contains several fields, including the block index, timestamp, data, and hash values.

The program defines a function called calculate_hash, which takes a pointer to a block structure as an argument, and calculates the hash value of the block using a simple hashing algorithm.

The program initializes a new block with some test data.

The program sets up a loop to mine the block, using OpenMP to parallelize the mining process.

The program initializes a nonce value to 0, and increments it in each iteration of the loop.

The program calculates the hash of the block using the current nonce value, and checks if the hash meets the required difficulty level. If the hash does not meet the difficulty level, the loop continues with the next nonce value.

If the hash meets the difficulty level, the program prints a message indicating that the block has been successfully mined, and sets a flag to indicate that mining is complete.

The program continues looping until either the maximum nonce value is reached or mining is complete.

Once mining is complete, the program calculates the time taken for block mining and prints it to the console.

Note that the program uses OpenMP to parallelize the mining process by splitting the loop iterations among multiple threads. The private and shared clauses are used to specify which variables are private to each thread and which variables are shared among all threads. The critical pragma is used to ensure that only one thread can print the "Block mined" message at a time.

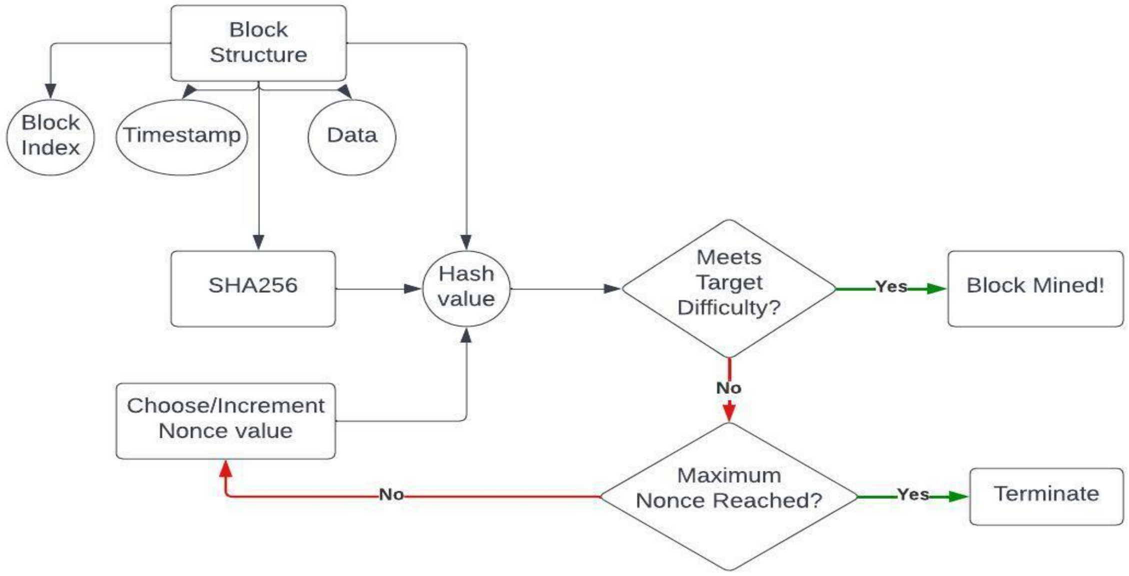


Fig. 3.4.10.1 : Blockchain Mining Flowchart

RESULTS

Fig. 4.1.1 : SHA256 Serial Implementation

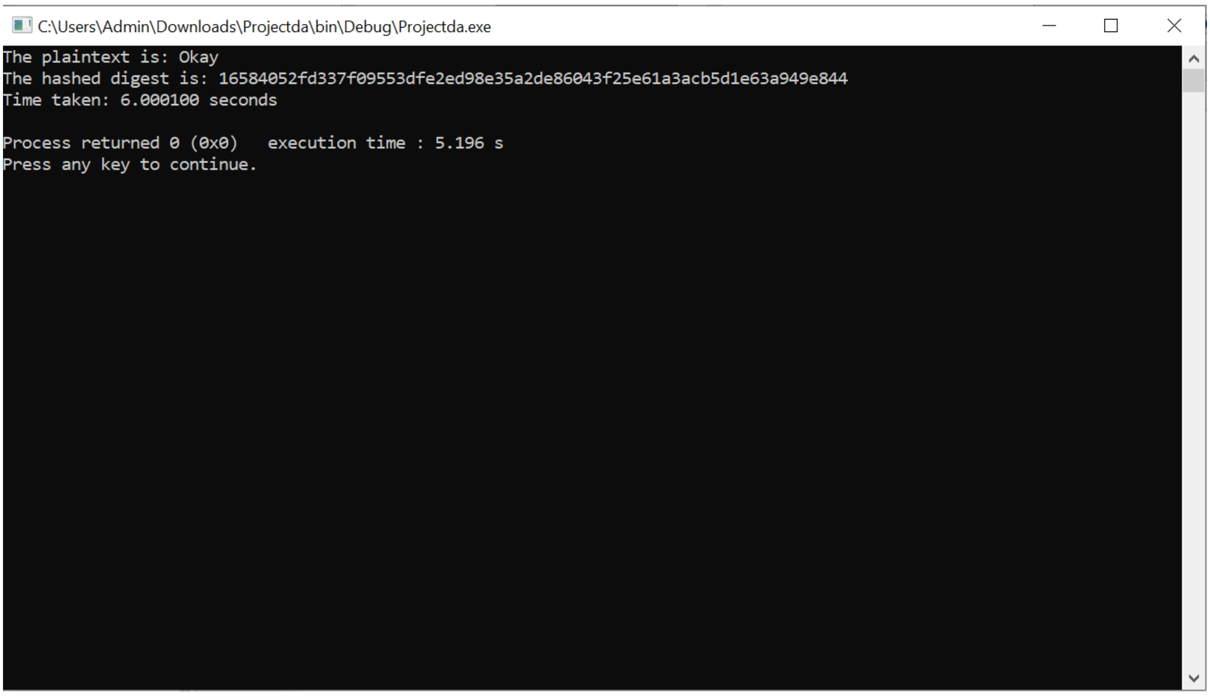


Fig. 4.1.2 : SHA256 Parallel Implementation

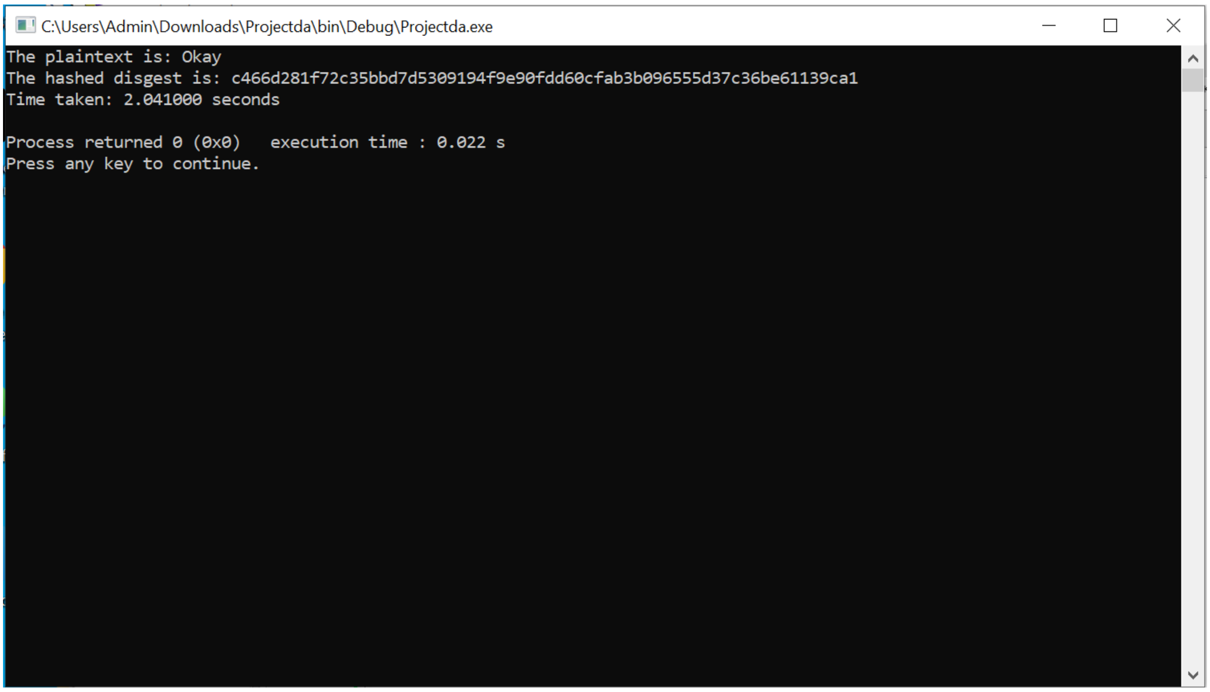


Fig. 4.1.3 : Blockchain Mining Serial Implementation

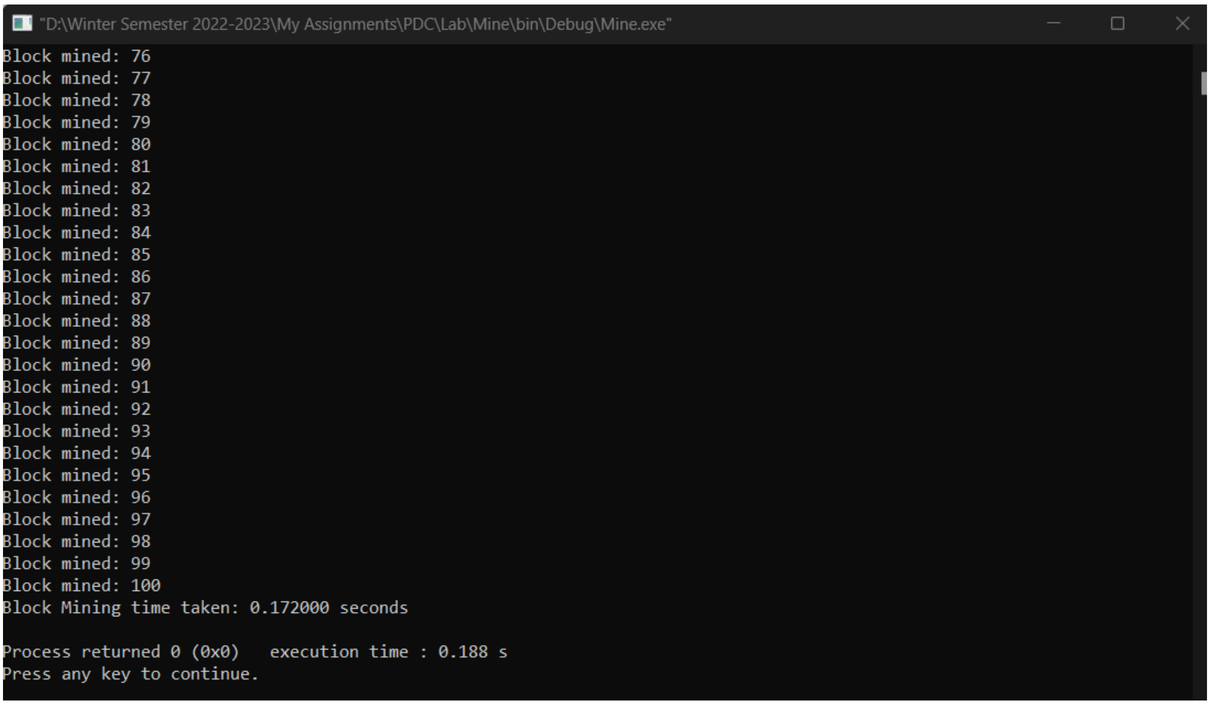
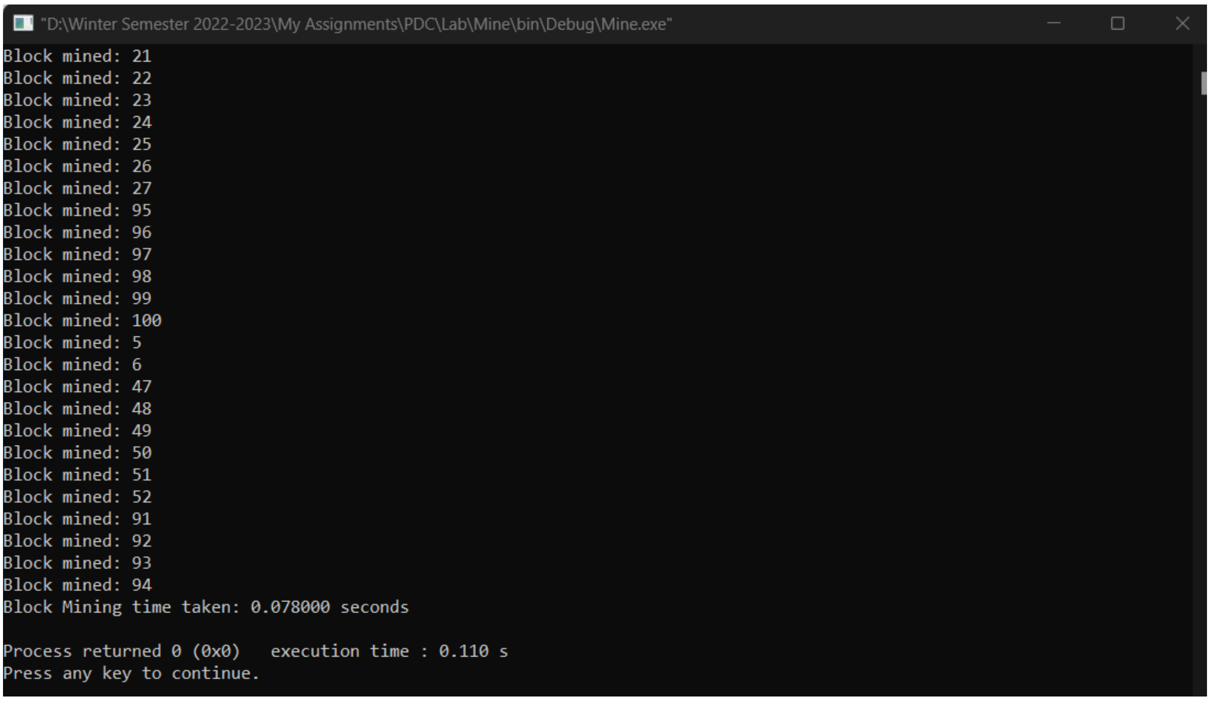


Fig. 4.1.4 : Blockchain Mining Parallel Implementation



Comparative Statistical Analysis

SHA256	Serial	Parallel
Time taken to hash	~6 s	2.04 s
Execution Time	5.196 s	0.022 s

Table 4.2.1 : SHA256 analysis for an arbitrary-length input

Blockchain Mining	Serial	Parallel
Mining Time	0.172 s	0.078 s
Execution Time	0.188 s	0.110 s

Table 4.2.2 : Blockchain Mining analysis for an arbitrary difficulty level

Length of String	Serial	Parallel
4	~6 s	2.04 s
6	6.81 s	2.22 s
10	7.42 s	3.52 s
18	10.11 s	6.81 s
20	~11 s	6.90 s
25	13.62 s	8.11 s
30	~16 s	10.20 s

Table 4.2.3 : SHA256 Analysis for variable input length – Serial v/s Parallel

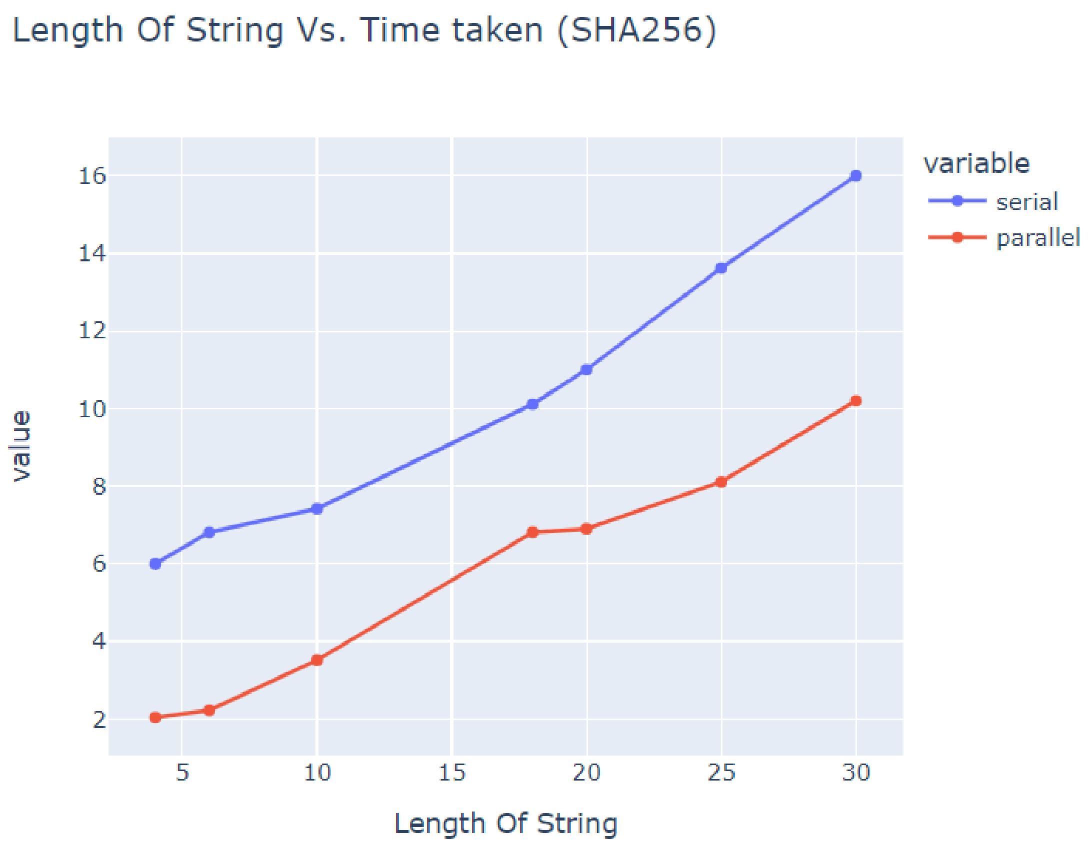


Fig. 4.2.1 : Plot for Length of String v/s Hashing Time.

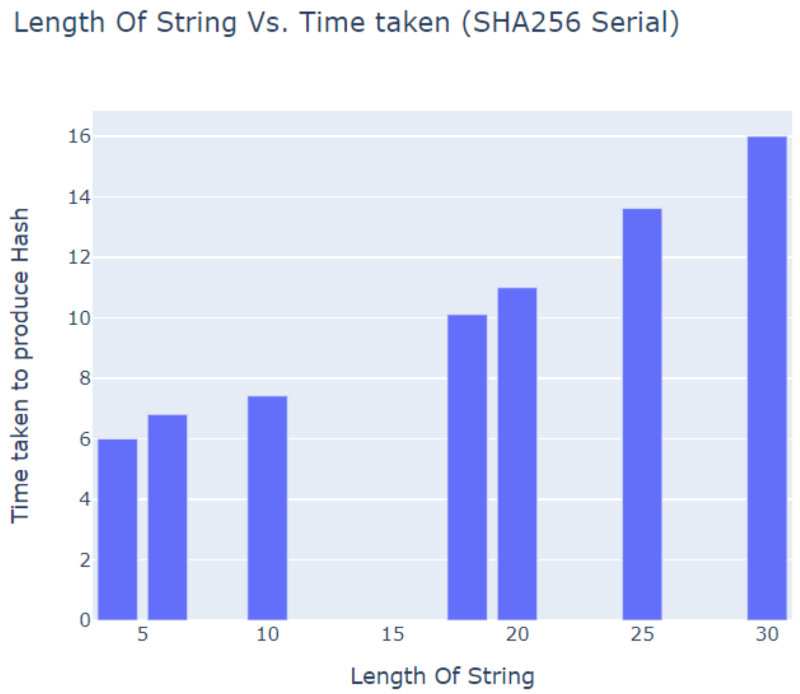


Fig. 4.2.2 : Bar plot for Serial Analysis – Length of String v/s Hashing Time.

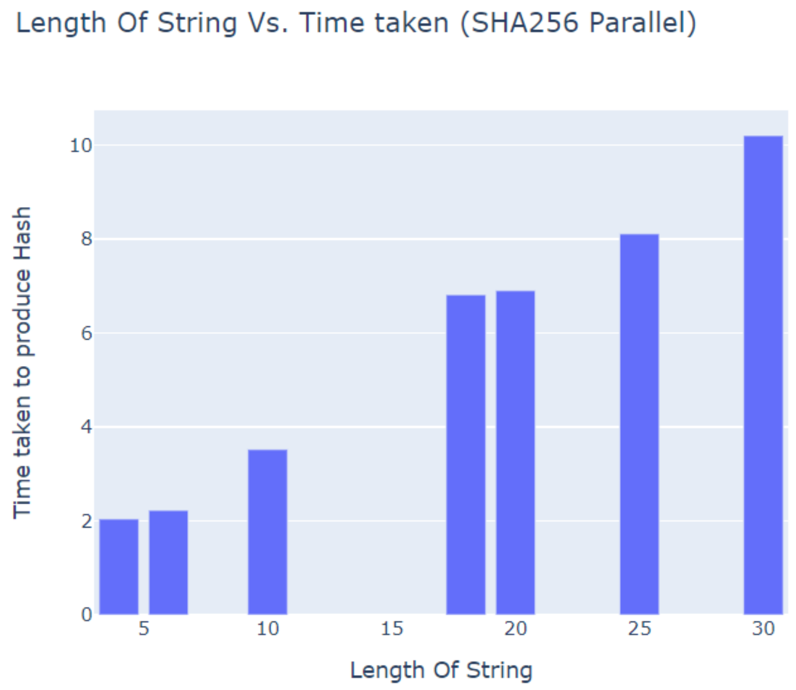


Fig. 4.2.3 : Bar plot for Parallel Analysis – Length of String v/s Hashing Time.

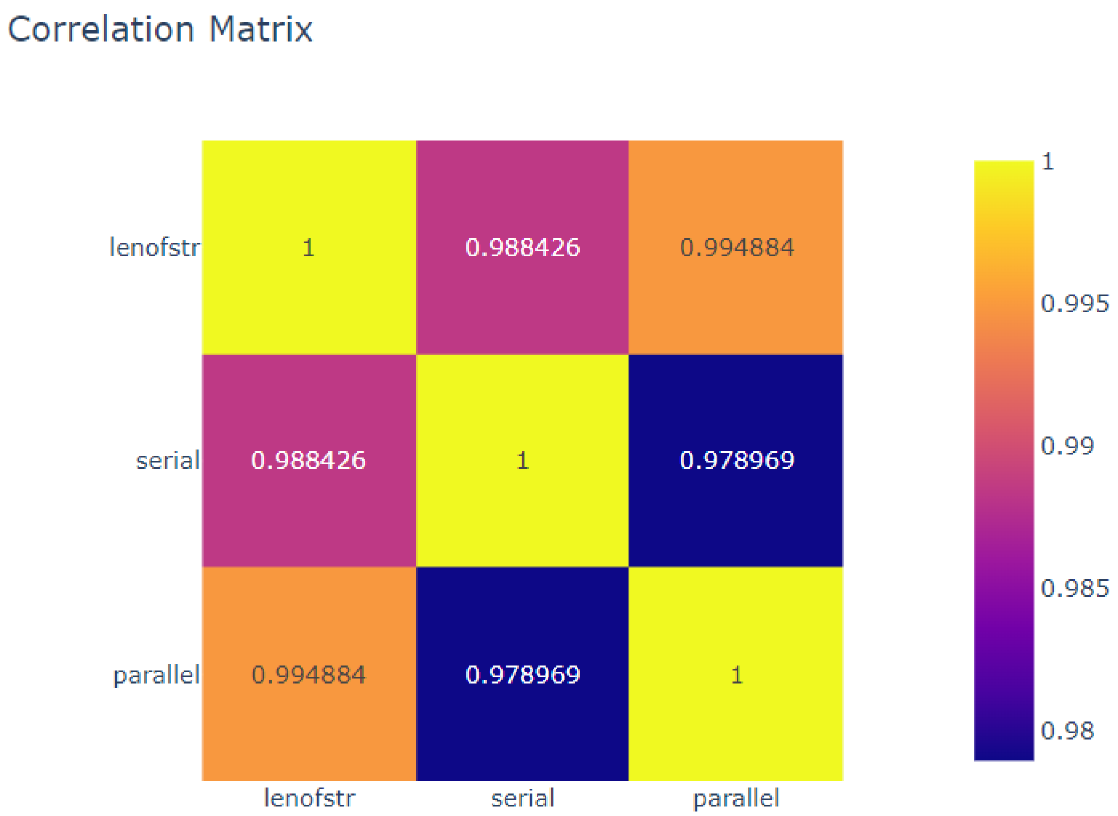


Fig. 4.2.4 : Heat-map for correlation analysis between Length of String and Serial v/s Parallel Hashing time.

Difficulty Level	Serial	Parallel
4	0.172 s	0.078 s
6	0.252 s	0.101 s
10	1.002 s	0.442 s
14	1.807 s	1.032 s
20	2.224 s	1.528 s

Table 4.2.4 : Blockchain Mining Analysis for variable difficulty level – Serial v/s Parallel.

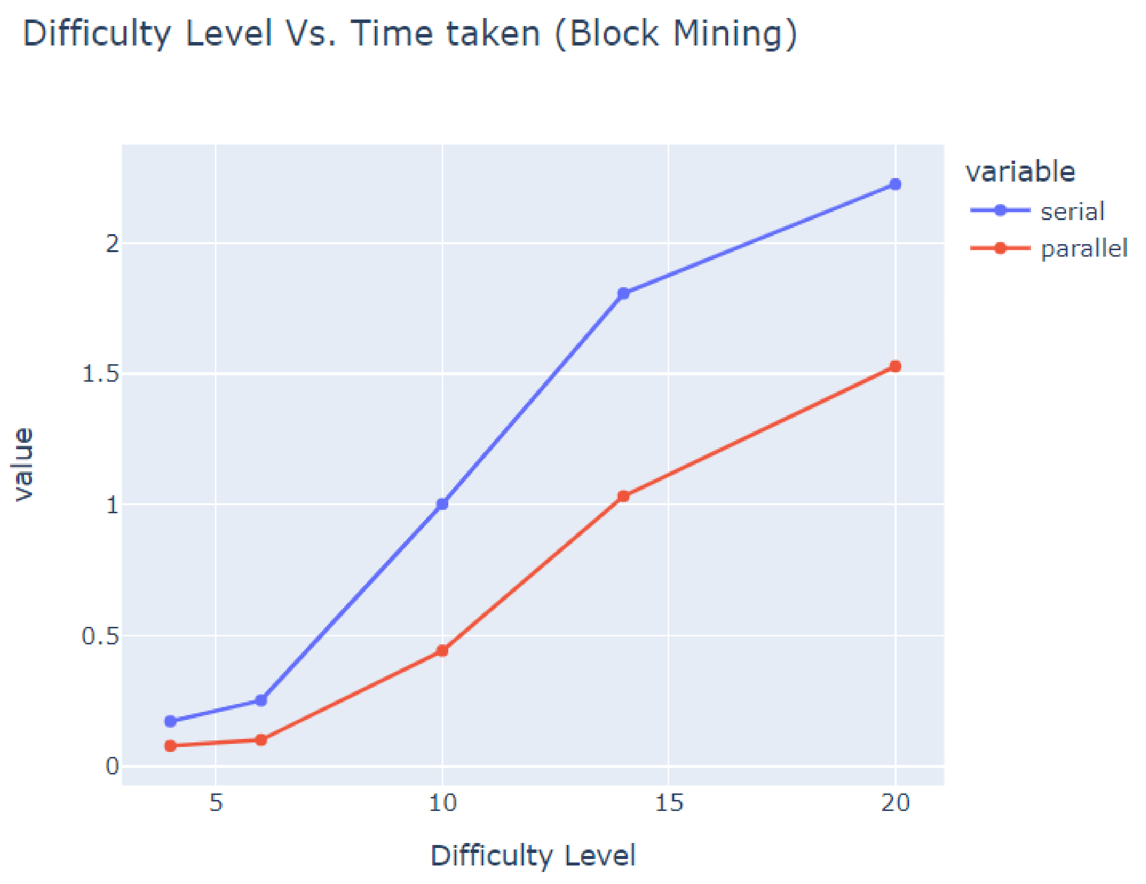


Fig. 4.2.5 : Plot for Difficulty level of blockchain v/s Mining Time.

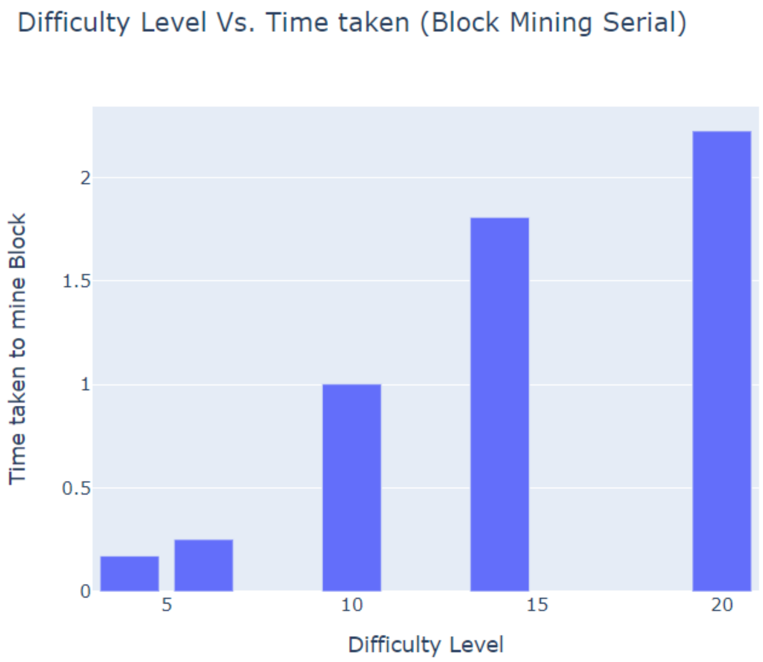


Fig. 4.2.6 : Bar plot for Serial Analysis – Difficulty level v/s Mining Time.

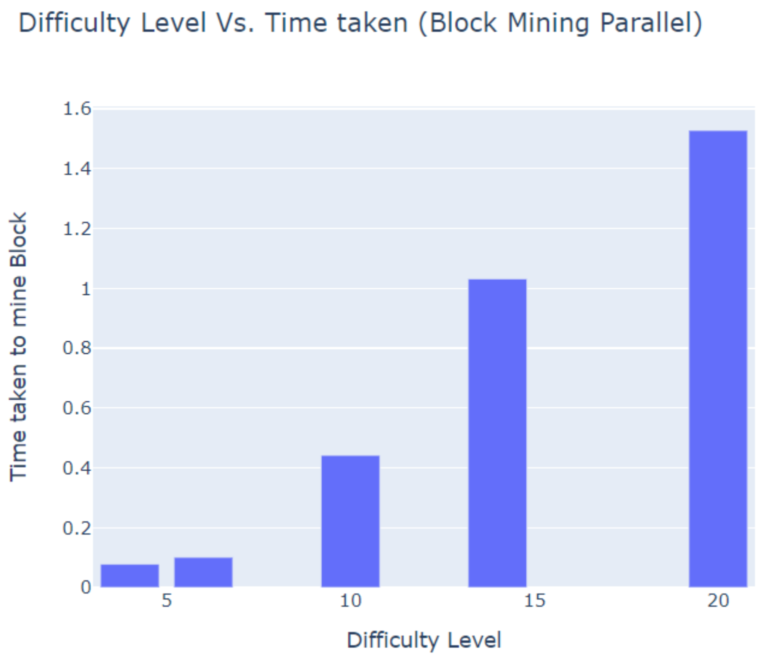


Fig. 4.2.7 : Bar plot for Parallel Analysis – Difficulty level v/s Mining Time.

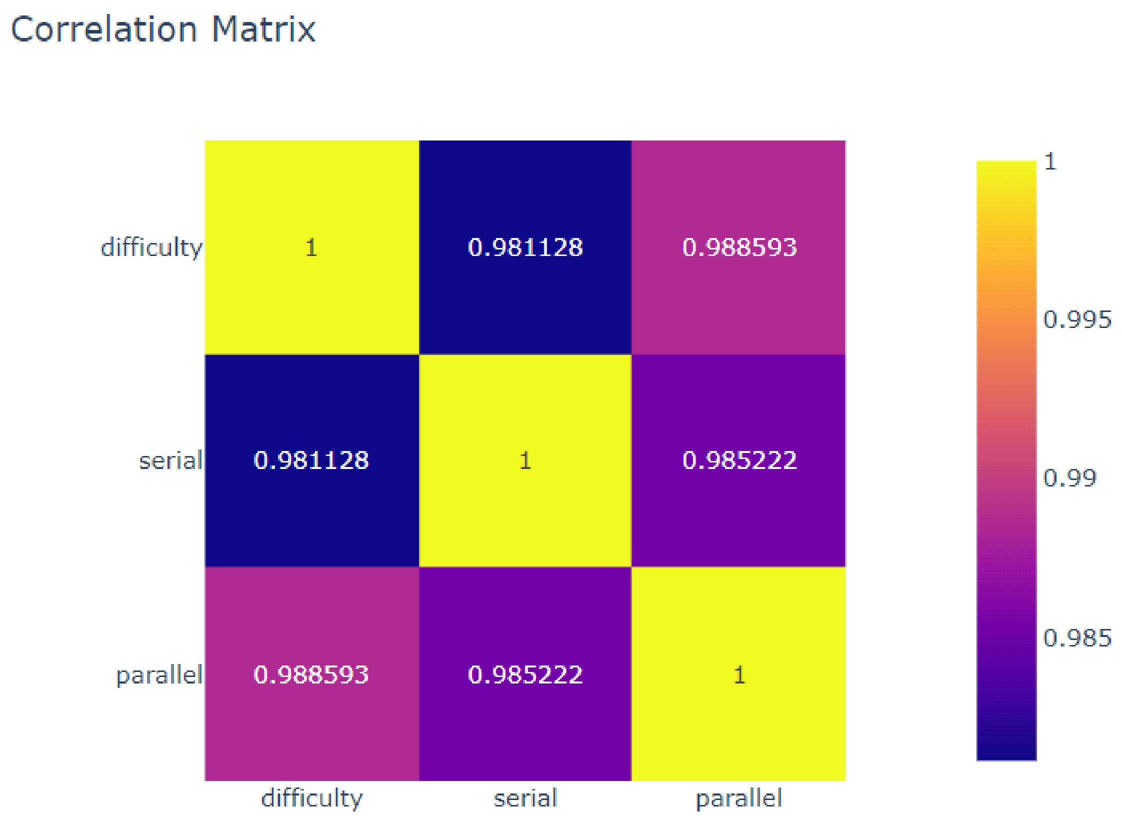


Fig. 4.2.4 : Heat-map for correlation analysis between Difficulty level and Serial v/s Parallel Mining Time.

CONCLUSION AND FUTURE WORK

We have implemented parallelization of SHA256 and Blockchain Mining processes. The results testify the fact that parallelized version has lesser latency as compared to serialized version. The execution times differ greatly given the scale of implementation.

In conclusion, the implementation of parallelization techniques in the SHA256 algorithm and Blockchain mining has proven to be a promising approach for improving the efficiency and scalability of these systems. By leveraging the power of parallel processing, it is possible to significantly reduce the time and computational resources required for mining and verifying transactions, thereby enhancing the overall performance of the Blockchain network. While there are still some challenges to be addressed, such as ensuring the security and integrity of the system, the use of parallelization techniques holds great potential for the future of Blockchain technology. As the demand for faster and more efficient mining and transaction processing continues to grow, it is likely that parallelization will become an increasingly important area of research in the field of Blockchain.

APPENDICES/ANNEXURES

I. Hardware and software specifications of the experimental setup

RAM required : 2GB
CodeBlocks IDE version : 20.03
Number of cores in processor : 16
GPU : Nvidia Geforce GTX 1650TI

II. Performance Analysis Plots

Line chart : A line chart or line graph, also known as curve chart, is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.
Bar plot : A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally.
Heat Map and Correlation matrix : A heat map (or heatmap) is a data visualization technique that shows magnitude of a phenomenon as color in two dimensions. The variation in colour may be by hue or intensity, giving obvious visual cues to the reader about how the phenomenon is clustered or varies over space.

III. SHA256 Algorithm Per-round computation pre-defined formulae

$$Ch(e,f,g) = (e \wedge f) \oplus (\neg e \wedge g)$$

$$Maj(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

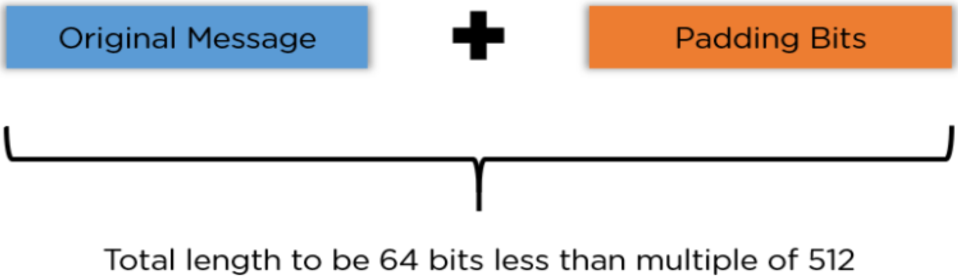
$$\sum\nolimits_1^{\{256\}}(e) = ROTR^6(e) \oplus ROTR^{11}(e) \oplus ROTR^{25}(e)$$

$$\sum\nolimits_1^{\{256\}}(a) = ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$$

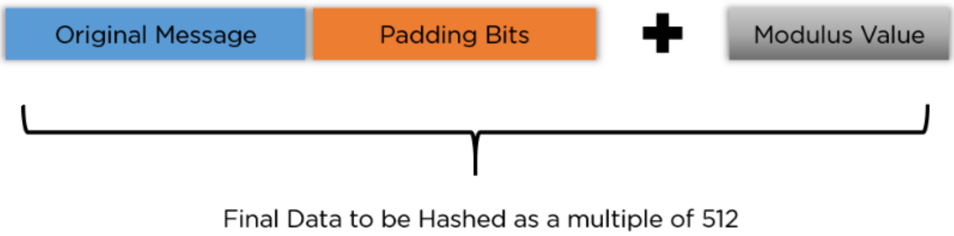
$K_t^{\{256\}}$= A set constant in the Bitcoin code (different for each iteration)

IV. SHA256 Algorithm Steps

Padding Bits
It adds some extra bits to the message, such that the length is exactly 64 bits short of a multiple of 512. During the addition, the first bit should be one, and the rest of it should be filled with zeroes.



Padding Length
You can add 64 bits of data now to make the final plaintext a multiple of 512. You can calculate these 64 bits of characters by applying the modulus to your original cleartext without the padding.



Initialising the Buffers:
You need to initialize the default values for eight buffers to be used in the rounds – a,b,c,d,e,f,g, and h.
You also need to store 64 different keys in an array, ranging from K[0] to K[63].

REFERENCES

[1] S. S. Hazari and Q. H. Mahmoud, "A Parallel Proof of Work to Improve Transaction Speed and Scalability in Blockchain Systems," 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2019, pp. 0916-0921, doi: 10.1109/CCWC.2019.8666535.

[2] S. Baheti, P. S. Anjana, S. Peri, and Y. Simmhan, “DiPETrans: A framework for distributed parallel execution of transactions of blocks in blockchains,” Concurrency and Computation: Practice and Experience, vol. 34, no. 10, Jan. 2022, doi: https://doi.org/10.1002/cpe.6804.

[3] M. Tote, A. Kumar, M. Mahankal, S. Khadse, and V. Uprikar, “Issue 5 www.jetir.org (ISSN-2349-5162),” JETIRBV06134 Journal of Emerging Technologies and Innovative Research, vol. 6, 2019 [Online]. Available: https://www.jetir.org/papers/JETIRBV06134.pdf

[4] M. J. Amiri, D. Agrawal, and A. El Abbadi, “ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems,” IEEE Xplore, Jul. 01, 2019. https://ieeexplore.ieee.org/document/8885058.

[5] R. Wu, X. Zhang, M. Wang and L. Wang, "A High-Performance Parallel Hardware Architecture of SHA-256 Hash in ASIC," 2020 22nd International Conference on Advanced Communication Technology (ICACT), Phoenix Park, Korea (South), 2020, pp. 1242-1247, doi: 10.23919/ICACT48636.2020.9061457.

[6] P. S. Anjana, “Efficient parallel execution of block transactions in blockchain,” Proceedings of the 22nd International Middleware Conference: Doctoral Symposium, Dec. 2021, doi: https://doi.org/10.1145/3491087.3493676.

[7] Z. Wang, X. Dong, Y. Kang, and H. Chen, “Parallel SHA-256 on SW26010 many-core processor for hashing of multiple messages,” The Journal of Supercomputing, vol. 79, no. 2, pp. 2332–2355, Aug. 2022, doi: https://doi.org/10.1007/s11227-022-04750-7.