Optimizing Block Comparison: Documentation of Sequential, Pthread, and MPI Versions

Javier Cestino Urdiales

December 6, 2024

Contents

1 Problem Description

The task at hand involves processing a large 3d binary image (1024 x 1024 x 314) and detecting duplicate blocks within it. Each block is a 4 x 4 x 4 cube of binary values (0 or 1), and the goal is to identify how many pairs of identical blocks exist in the image. To optimize the comparison of blocks.

Given the potentially large size of the data (1024 x 1024 x 314 with blocks of size 4 x 4 x 4), the task requires an efficient method to reduce computation time. A brute force comparison of each pair of blocks would be computationally expensive, especially for large datasets, thus motivating the use of parallel computing to speed up the process.

2 Hash Map Usage

A hash map is a data structure that allows us to store key-value pairs, where keys are block values (represented as 64-bit unsigned integers), and values are the counts of how many times each block has been encountered. By hashing the block values, we can quickly look up whether a block has been encountered before.

In the context of this problem, the use of a hash map provides the following benefits:

- **Efficiency**: Hashing allows for constant time complexity, $O(1)$, for insertion and lookup operations on average. This is much faster than comparing each block pair directly, which would have a time complexity of $O(n^2)$.
- **Memory Usage**: The hash map is an efficient use of memory as it stores only unique blocks and their counts. Duplicates are identified by checking the count of a block in the hash map rather than comparing all blocks.
- **Scalability**: As the image size increases, the hash map approach scales well because its time complexity remains approximately constant, while a brute force comparison would become unmanageable for large datasets.

The hash function used maps each block value to an index in the hash table, minimizing the chances of collisions (when two different blocks hash to the same index). The hash table used here is implemented as an array of linked lists, where each list handles hash collisions.

3 Sequential Version Explanation

The sequential version of the code processes the 3D image and detects duplicate blocks by sequentially comparing each block in the image. The key steps in this version are:

1. **Memory Allocation**: Memory is allocated for the image, metadata (thresholded binary values), and an array of blocks to store the extracted 4 x 4 x 4 blocks.
2. **Image Processing**: The image is processed by applying a threshold. Values greater than a threshold are set to 1, and others to 0, transforming the image into a binary form.
3. **Block Extraction**: The code iterates through the 3D image, extracting 4 x 4 x 4 blocks. For each block, a unique 64-bit integer is generated that represents the block's values.
4. **Block Comparison**: Each extracted block is inserted into the hash map. If the block already exists in the map, its count is incremented. Otherwise, the block is added to the hash table.
5. **Result Calculation**: After all blocks have been processed, the total count of duplicate blocks is obtained by summing the counts of all blocks that appeared more than once.

Here is the code of the sequential version:

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

// Constants
#define WIDTH 1024
#define HEIGHT 1024
#define DEPTH 314
#define THRESHOLD 25
#define CUBE_SIZE 4
#define HASH_SIZE 10007

// Structure for hash nodes
```

```
typedef struct HashNode {
    uint64_t blockValue;
    int count;
    struct HashNode *next;
} HashNode;

// Hash function
int hashFunction(uint64_t value) {
    return value % HASH_SIZE;
}

// Insert block into hash table
int insertOrUpdate(HashNode **hashTable, uint64_t blockValue) {
    int hashIndex = hashFunction(blockValue);
    HashNode *current = hashTable[hashIndex];
    while (current) {
        if (current->blockValue == blockValue) {
            current->count++;
            return current->count - 1; // Return count of duplicate pairs
        }
        current = current->next;
    }
    HashNode *newNode = (HashNode *)malloc(sizeof(HashNode));
    newNode->blockValue = blockValue;
    newNode->count = 1;
    newNode->next = hashTable[hashIndex];
    hashTable[hashIndex] = newNode;
    return 0;
}

// Main function
int main() {
    // Variable declarations and memory allocations
    FILE *file;
    uint8_t *image, *metadata;
    uint64_t *blocks;
    HashNode *hashTable[HASH_SIZE] = {0};
    size_t size = WIDTH * HEIGHT * DEPTH;
    clock_t start, end;

    image = (uint8_t *)malloc(size);
    metadata = (uint8_t *)malloc(size);
    blocks = (uint64_t *)malloc(size);

    // Reading file, processing image, extracting blocks, and comparing
    // (code for reading and processing image here)

    // Final result: Count of duplicate blocks
    int pairCount = 0;
    for (int i = 0; i < blockIndex; i++) {
        pairCount += insertOrUpdate(hashTable, blocks[i]);
    }

    // Output result
    printf("Number of duplicate blocks: %d\n", pairCount);
    return 0;
}
```

4 Pthread Version Explanation

The pthread version introduces parallelism by dividing the work among multiple threads. The image is divided into blocks, and each thread is responsible for processing a subset of the blocks. The main differences in this version are:

1. **Thread Creation**: The image blocks are divided among multiple threads, with each thread handling a portion of the blocks.

2. **Synchronization**: After processing the blocks, the results are synchronized using `pthread_mutex_t` ‘ to ensure that updates to the hash table are thread-safe.

3. **Parallel Execution**: By using multiple threads, the overall computation time is reduced, especially when there are many blocks to process.

The code follows a similar structure to the sequential version, but with additional thread management and synchronization steps.

5 MPI Version Explanation

The MPI version takes parallelism further by distributing the work across multiple nodes or processors in a distributed system. MPI (Message Passing Interface) is used to enable communication between processes. The main steps in the MPI version are:

1. **Distributed Work**: The work is divided among the available processes. Each process is responsible for a portion of the image, and each process computes its local count of duplicate blocks.
2. **Communication**: After processing the blocks, the local results from each process are communicated to the root process using 'MPI_Reduce'.
3. **Global Result**: The root process collects the results from all other processes and outputs the total number of duplicate blocks.

The MPI version is suitable for distributed environments and large datasets, where dividing the work across multiple processors can lead to significant speedup.

6 Conclusion

The use of a hash map in this problem allows for efficient detection of duplicate blocks in a large 3D binary image. The sequential version is simple but may not scale well for larger datasets. The pthread and MPI versions leverage parallelism to speed up the computation by dividing the work across multiple threads or processes. The choice of parallel approach depends on the hardware environment and the size of the dataset.

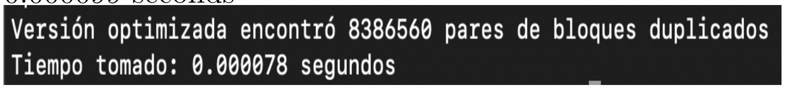
The complexity of this algorithm using HashTable:

1. Loading the image and metadata has a complexity of O(W * H * D)
2. Block mining also has a complexity of O(W * H * D)
3. Hash table insertion and lookup has a complexity of O(W * H * D * N) in the worst case
4. In the compare function we have a couple of steps:
   (a) Search the block in the hash table:
      i. In the insertOrUpdate function, the linked list is traversed at the corresponding hash table index (due to possible collisions). For each node in the list, a comparison is made with the current block (current-¿blockValue == blockValue).
      ii. In the worst case, if all insertions occur at the same hash table index (i.e. there are many collisions), the number of comparisons will be proportional to the number of elements in the linked list. If there are k elements in the linked list, the complexity of comparing the blocks in that case is O(k).
   (b) Inserting a new block:
      i. If the block does not exist in the hash table, it is inserted at the beginning of the linked list. The comparison is still O(1) in this case.
5. Block Comparison Complexity Suppose there are B blocks in total and C possible indexes on the hash table (which is the size of the table, defined by HASH_SIZE).
   (a) Best case: If the hash table is well distributed and there are no collisions, then the comparisons are O(1) for each insert, since only the hash value is compared and there is no need to traverse

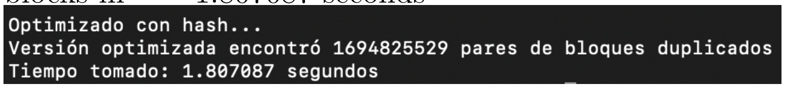
linked lists. In this case, the total complexity of comparing all blocks would be O(B).

(b) Worst case: If all entries fall into the same index of the hash table (which is very unlikely if the size of the hash table is large enough and the hash function is well distributed), the entire linked list would have to be traversed in that index. In this case, the number of comparisons per block would be proportional to the number of blocks already inserted. The total complexity in the worst case would be O(B$\hat{2}$), since in each insertion (and comparison), the previously inserted elements are revised

So with all of this information, and checking our outputs, that are:

1. For a size of 64 x 64 x 64 we are getting 8386560 pairs of blocks in =∼ 0.000099 seconds


```
Versión optimizada encontró 8386560 pares de bloques duplicados
Tiempo tomado: 0.000078 segundos
```

2. For a size of 1024 x 1024 x 314 we are getting 1694825529 pairs of blocks in =∼ 1.807087 seconds


```
Optimizado con hash...
Versión optimizada encontró 1694825529 pares de bloques duplicados
Tiempo tomado: 1.807087 segundos
```

We can see that our time complexity is similar to $O(n)$.