CHAPTER IV

A MEMORY-EFFICIENT PARALLEL ALGORITHM
FOR HUFFMAN DECODING

4.1. Introduction

As the computing power available increases, the quest for more and more power also keeps increasing. Numerous applications, which we have discussed in the last chapter, are in need of memory efficient fast algorithms. Processors have brought flexibility and programmability to the computational world. The emerging Digital Signal Processors (DSP) are becoming fast in order to run the state-of-the-art applications like Audio, Imaging, Video and Decoders [20,21,22]. Even though the emerging DSPs have sufficient MIPS (Million Instructions per Second) for running above-mentioned applications, it is imperative that the applications consume less MIPS and memory to provide cost-effective solutions to the end users. Though the DSP architectures are optimized for signal processing applications, they are not so in case of search algorithms. But, Huffman encoding/decoding, which uses search algorithms, has become one of essential components of the compression standards [29]. Hence it is essential that the Huffman encoding/decoding should be efficiently implemented with respect to memory. The complexity of implementation lies in fast search of the symbol encoded from the bit-stream without consuming large memory. Keeping these in view, we extend our Parallel Huffman Decoding Algorithm, presented in the previous chapter, to a memory efficient Huffman decoding algorithm on CREW PRAM model, which requires only $\lceil \frac{3n}{2} \rceil + \lceil n/2 \log n \rceil + 1$ memory space for
storing input and takes $O (\log N+1 (n+1))$ running time where $n$ is the number of symbols in a Huffman tree.

### 4.2. Data Structure

Consider a Huffman tree $T$ with $n$ symbols (representing messages). The leaves are labeled as $m_1, m_2...m_n$ from left to right. The weight of a symbol is defined to be $2^{d_p}$, where $d$ is the depth (height) of the Huffman Tree and $p$ is the level of the symbol. Let $W_i$ be the weight of the symbol $m_i$ for $i=1,2...n$. Define $cum_1 = W_1$ and $cum_i= cum_{i-1} + W_i$ for $i = 2,3...n$. The Huffman tree is shown in Figure 4.1. and the values of $W_i$, $cum_i$, $m_i$, $i= 1,2...n$ are shown in Table 4.1. Note that $W_i$ can be obtained from $W_i = cum_i - cum_{i-1}$ for $i = 2$ to $n$ and $W_1 = cum_1$. Therefore, the array $W$ in Table-1 can be omitted. The required memory space, now, reduces to $2n$. Now, we concentrate on decreasing the memory space needed by array $cum$.

Let $W_i= W_{2i-1}+W_{2i}$ for $i = 1,2...\lceil n/2 \rceil$. Note that $W_{\lfloor n/2 \rfloor} = W_n$ when $n$ is odd. Let $CUM_1 = W_1$ and $CUM_i = CUM_{i-1}+ W_i$, $i = 2$ to $\lceil n/2 \rceil$. A bit $b_i$ is used to indicate if $W_{2i-1} \leq W_{2i}$ or not. If so, $b_i$ is zero, for $i = 1$ to $\lceil n/2 \rceil$. It is, further, to be noted that it is unnecessary to store array $W$ either, as it can be computed from the elements of array $cum$. Hence, Table 4.1. can be condensed to Table 4.2. with the changes discussed in the previous section. So the data structure that helps to achieve memory efficiency with $\lceil 3n/2 \rceil + \lceil n/2 \log n \rceil + 1$ spaces is the one akin to Table 4.2.
Figure 4.1. *A Huffman Tree*
<table>
<thead>
<tr>
<th>INDEX</th>
<th>$m_i$</th>
<th>$W_i$</th>
<th>CUM$_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$m_1$</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>$m_2$</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>$m_3$</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>$m_4$</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>$m_5$</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>$m_6$</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>$m_7$</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>$m_8$</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>$m_9$</td>
<td>4</td>
<td>32</td>
</tr>
</tbody>
</table>

**Table 4.1.** The values of $m_i$, $W_i$, CUM$_i$
Table 4.2. *The values of CUM*<sub>i</sub>, *i*, *b*<sub>i</sub>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUM&lt;sub&gt;i&lt;/sub&gt;</td>
<td>8</td>
<td>13</td>
<td>16</td>
<td>28</td>
<td>32</td>
</tr>
<tr>
<td><em>b</em>&lt;sub&gt;i&lt;/sub&gt;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
The array $CUM$, the cumulative function, has distinct elements increasing in order. In our algorithm we compute a value $t$ and search in $CUM$. Therefore, our problem in question reduces to a problem of searching an element $t$ in a set of distinct and sorted elements. This prompts the use of $(N+1)$-ary search (Parallel version of BINARY SEARCH), where $N$ is the number of processors.

4.3. Memory Efficient Parallel Algorithm

In this section, a memory efficient parallel algorithm on CREW PRAM computational model is presented.

**ALGORITHM**

**Input:** The arrays $m$ (the set of messages), $CUM$ and $b$ of a Huffman tree $T$ with height $h$ and a binary codeword $c$.

**Output:** The corresponding symbol $m_k$ of $c$.

**METHOD:**

**Step 1:** Compute $t \leftarrow (c+1) \times 2^d$, where $d$ is the number of binary digits in $c$.

**Step 2:** {Find $CUM_k$ such that $CUM_{k-1} < t \leq CUM_k$}

2.1) $q \leftarrow 1$

2.2) $r \leftarrow n$

2.3) $k \leftarrow 0$

2.4) $\text{stages} \leftarrow \lceil \log(n+1) / \log(N+1) \rceil$

2.5) While ($q \leq r$ and $k = 0$) do

   $j_0 \leftarrow q -1$

   for $i = 1$ to $N$ do in Parallel
Processor Pi compares t to CUMji and decides on the part to be retained.

if $ji \leq r$
then if $\text{CUM}_{s} = t$
then $k \leftarrow ji$
else if $\text{CUM}_{s} > t$
then $\text{flag}_{i} \leftarrow \text{left}$
else $\text{flag}_{i} \leftarrow \text{right}$
end if
end if
else
$ji \leftarrow r+1$
$\text{flag}_{i} \leftarrow \text{left}$
end if

{The indices of the subsequence to be searched in the next iteration are computed}

if $\text{flag}_{i} \neq \text{flag}_{i-1}$
then $q \leftarrow ji-1 + 1$
$r \leftarrow ji - 1$
end if

if ($i = N$ and $\text{flag}_{i} \neq \text{flag}_{i+1}$)
then $q \leftarrow ji + 1$
end if

Step 3: if (k = 0)

then k ← q

endif

Step 4: x ← CUMₖ - CUMₖ₋₁

Step 5: Decompose x into x₁ and x₂ such that x = x₁ + x₂, x₁ = 2ₑ₁ and x₂ = 2ₑ₂, for some non-negative integers e₁ and e₂ and assume without loss of generality that e₁ ≤ e₂.

Step 6: Use bₖ, x₁, x₂ to determine Wₐ and Wᵦ which are the respective weights of m₂ₖ₋₁ and m₂ₖ.

Step 7: if t = CUMₖ and Wᵦ = 2ʰ₋ᵈ

then m₂ₖ is the corresponding message of the codeword c.

Step 8: if t = CUMₖ - Wᵦ and Wₐ = 2ʰ₋ᵈ

then m₂ₖ₋₁ is the corresponding message of the codeword c else
c is not a codeword of T.

4.4. Illustration

We illustrate our algorithm with the following examples.
4.4.1. Example 1:

Let \( c = 01101 \).

In step-1, we compute \( t = (13+1) \cdot 2^{5-5} = 14 \).

In step-2, we find \( k = 3 \), as \( 13 < 14 < 16 \), having \( N \) be 2.

In step-3, \( x = 16 - 13 = 3 \).

In step-4, \( x_1 = 2^0; x_2 = 2^1 \).

In step-5, \( W_a = 1; W_b = 2; b_3 = 0 \).

In step-6, Since \( t \neq \text{CUM}_k \), fails.

In step-7, Since \( t = 16 - 2 \) and \( W_a = 2^{5-5}, 2k-1 = 5 \) and, hence, \( m_5 \) is the corresponding message of the codeword \( c \).

4.4.2. Example 2:

Let \( c = 1011 \).

In step-1, we compute \( t = (11+1) \cdot 2^{5-4} = 24 \).

In step-2, we find \( k = 4 \), as \( 16 < 24 < 28 \), having \( N \) be 2.

In step-3, \( x = 28 - 16 = 12 \).

In step-4, \( x_1 = 2^2; x_2 = 2^3 \).

In step-5, \( W_a = 8; W_b = 4, \) as \( b_4 = 1 \).

In step-6, Since \( t \neq \text{CUM}_k \), fails.

In step-7, Since \( t = 28 - 4 \) and \( W_a \neq 2^{5-4}, c \) is not a codeword of \( T \).

4.5. DISCUSSION

The algorithm presented in this chapter, employs \((N+1)\)-ary search, the extend version of binary search in step- 2 and it takes \( O (\log_{N+1}(n+1)) \) time. Since
the remaining steps performed in constant time, the overall time complexity of this algorithm takes \( O(\log^{N+1}(n+1)) \), where \( 1 < N \leq n \). The main significance of this algorithm is its memory efficiency. It requires only \( \lceil 3n/2 \rceil + \lceil n/2 \log n \rceil + 1 \) memory space for storing input.

Computational requirement have always been on the increase. As the processors become more powerful, we are interested in getting the machine to tackle more complex problems. Most of today's architectures use pipelining at the instruction level to increase the throughput. In our study, we have also understood that the Linear Array with Reconfigurable Pipelined Bus System (LARPBS) model is an efficient parallel computational model, which can be dynamically reconfigured into independent sub-systems of smaller sizes, which can be used to solve sub-problems simultaneously. As a matter of fact, an LARPBS is no less powerful than a completely connected electronic network (which has unbounded degree) in terms of interprocessor communications and global aggregations, and is more powerful than any bounded-degree electronic networks. Hence, in order to improve the speed, our attention has been focused on developing a Huffman decoding algorithm on this new computational model i.e. LARPBS model, which is described in the following chapter.