CHAPTER V

HUFFMAN DECODING ALGORITHM ON LARPBS MODEL

5.1. Introduction

With ever increasing applications of parallel computers, investigators are always trying to design new algorithms which would fully exploit the new variety of hardware with respect to both speed and the magnitude of the problems that can be handled. Many algorithms with inherent parallelism have a higher computational complexity than the best sequential counterpart. Hence, implementing an inappropriate algorithm wastes parallel computer resources. So, it is important to design a parallel algorithm that makes good use of the target architecture. But the current parallel processing systems require interconnections that can provide high bandwidth so as to satisfy communication demands of the parallel system. One of the main motivations for study and implementation of optical interconnect is to overcome the bottlenecks that electrical data buses produce due to their low bandwidth. High processing speed of current technology has over-passed the physical limitations of electrical interconnection. To overcome these problems, optical technologies has been implemented.

Recent advances in optical and opto-electronic technologies indicate that optical interconnection can be used effectively in massively parallel computing systems involving electronic processors. Several different opto-electronic parallel computing models have been proposed in the literature. There are two related opto-electronic models based on the idea of dynamically reconfigurable optical
buses, namely the Array with Reconfigurable Optical buses (AROB) and the Linear Array with Reconfigurable Pipelined Bus Systems (LARPBS). An excellent overview of these models is given in the introduction chapter. These models have opened up new challenges in algorithm design. Several algorithms from different domains have been designed for LARPBS model in the recent years. In order to make a significant contribution towards the implementation of this model, we have designed an $O(1)$-time parallel Huffman decoding algorithm using single side growing Huffman tree as a memory efficient data structure, which requires $(n + d)*(\log_2 n)$-bit memory space, where $n$ is the number of source symbols and $d$ is the depth of the Huffman tree.

It is of importance to mention that a Linear Array with reconfigurable pipelined Bus System is stronger than any other PRAM model with respect to implementability. Owing to its high communication bandwidth, support of versatile communication patterns and reconfigurability many problems can be solved more efficiently on LARPBS model than on a PRAM model. Recently Li et al., [43] have presented simulation techniques between the CREW PRAM and the LARPBS model. They have shown that each step of an algorithm on the CREW PRAM can be simulated on the LARPBS in $O(\log n)$ time, where $n$ is the number of processors involved. Hence a direct simulation of the algorithms by Lin and Chung [12] on the LARPBS will result in a degradation of the running time by a factor of $O(\log n)$. 

84
Our main achievement made in this work is that the algorithm in [5] can be simulated on the LARPBS without any degradation in running time, that is in O(1) time itself.

5.2. Preliminaries

5.2.1. Lemma-1: One-to-one communication, broadcasting, multicasting, multiple multicasting, binary prefix sum computation, and ordered compression can be done in O(1) bus cycles on the LARPBS model.

5.3. Data Structure

Let us consider the source symbols \{s_1, s_2...s_n\} with frequencies \{w_1, w_2...w_n\} for \(w_1 \geq w_2 \geq ... \geq w_n\), where the symbol \(s_i\) has frequency \(w_i\). Using the Huffman’s algorithm [4], a Huffman tree is obtained. Then the codeword \(c_i\) for \(1 \leq i \leq n\), which is a binary string, for symbol \(s_i\) can be determined by traversing the path from the root to left node associated with the symbol \(s_i\), where the left edge corresponding to ‘0’ and the right edge corresponding to 1. The level of the root is zero and the level of the other node is equal to summing up its parent’s level and one. Codeword length \(l_i\) for \(s_i\) can be known as the level of \(s_i\). Then the weighed external path length, \(\sum_{i=1}^{n} w_i l_i\), is minimum. From the set of codeword length \((l_1, l_2...l_n)\) in the Huffman tree, codeword assignment scheme has been found to generate a new set of code words for source symbols. Based on the codeword assignment scheme, single-side growing Huffman tree [12] is constructed in which it is very clear that if a half node associated with the symbol
sₐ in the original Huffman tree has codeword length ℓₐ the corresponding codeword is obtained by using the codeword scheme which has the same length ℓₐ.

In the single-side growing Huffman tree, let Iₖ be number of internal nodes at level k, and fₖ be the number of leaf nodes at level k, each leaf node with codeword length k. It satisfies the recurrence relation I₀ = 1 and Iₖ = (Iₖ₋₁ * 2) − fₖ. Let cᵢ' is the codeword. The logical address aᵢ for storing sᵢ in the symbol table is given by the following address assignment scheme:

\[ aᵢ = cᵢ' - Iᵢ + \sum_{k=2}^{L} f_{k-1} \text{ for } 1 ≤ i ≤ n \]

Let us consider a single-side growing Huffman tree with source symbols S = \{s₁, s₂, ..., s₈\} with the frequencies \{9, 7, 3, 3, 1, 1, 1, 1\} and lengths \{ℓ₁, ℓ₂, ℓ₃, ℓ₄, ℓ₅, ℓ₆, ℓ₇, ℓ₈\} = \{2, 2, 3, 3, 4, 4, 4, 4\} respectively. Using preceding codeword assignment scheme, we have \{c₁', c₂', ..., c₈'\} = \{11, 10, 011, 010, 0011, 0010, 0001, 0000\} and the single-side growing Huffman Tree is shown in Figure 5.1. Using address-assignment scheme, the symbol table corresponding to Figure 5.1 is shown in Table 5.1. The skip table is defined to be F = \{f₁, f₂, ..., fₘ\} where d is the depth of the single-side growing tree and fᵢ for 1 ≤ i ≤ d which is shown in Table 5.2.

In addition to that in our algorithm, we also need to keep Iᵢ, for 1 ≤ i ≤ d, and hence an internal table I = (I₁, I₂, ..., Iₘ) is built. We also need prefix-skip table P = (0, f₁, f₁ + f₂, ..., \(\sum_{i=1}^{d-1} fᵢ\)), which can be computed from the skip table F = (f₁, f₂, ..., fₘ). The Internal table and prefix–skip table corresponding to the Figure 5.1 is shown in Table 5.3 and Table 5.4 respectively.
Figure 5.1. Single-side growing Huffman tree
Table 5.1. *The Symbol Table*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_2$</td>
<td>$s_1$</td>
<td>$s_4$</td>
<td>$s_3$</td>
<td>$s_8$</td>
<td>$s_7$</td>
<td>$s_6$</td>
<td>$s_5$</td>
</tr>
</tbody>
</table>

Table 5.2. *The Skip Table*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
### Table 5.3. The Internal Table

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 5.4. The Prefix-Skip Table

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
In fact, these internal table and prefix-skip table can be obtained in the preprocessing step.

5.4. An O (l)-time parallel decoding Algorithm

In this section, we present a Huffman decoding algorithm on an LARPBS model using d processors. Since the decoding algorithm traverses the path in the single-side growing Huffman tree logically, the number of internal nodes of level $I_i$ can be computed which is described in section 2. So the internal table ($I_1$, $I_2$...$I_d$) is built. In addition, we need prefix-skip table $P$ which can be formed by using number of leaf nodes $f_k$, i.e., $P = (0, f_1, f_1 + f_2, \ldots, \sum_{i=1}^{d-1} f_i)$.

**ALGORITHM**

**Input** : The arrays $H = \{h_1, h_2 \ldots h_d\}$, $P = (0, f_1, f_1 + f_2, \ldots, \sum_{i=1}^{d-1} f_i)$, and $I=(I_1, I_2 \ldots I_d)$.

**Output** : The indices of the source symbols $S = \{s_1, s_2 \ldots s_n\}$ corresponding to $H$.

**Storage** : The array $H$ is stored in the first processor as a decimal number. Each array element $p[i]$ is stored in a processor $P_i$. Similarly each is stored in a processor $P_i$. The global variable and base are stored in the first processor of our model. We avoid the need for shared memory by broad-casting the appropriate values as and when needed. We also assume that each processor is capable of converting a decimal number to binary equivalent and vice versa.
METHOD:

**Step 1:** We broadcast base and H (its decimal form) to all the processors (from Processor-1).

**Step 2:** While \((base \leq r)\) do

Begin

(2.1) for \(i = 1\) to \(d-1\) do in parallel

Each processor \(P_i;\)

i. converts the decimal number \(H\) into binary equivalent and apportions a binary segment from positions \(base\) to \(base + i - 1\) (from left to right).

ii. finds the decimal equivalent, \(V_i\) of the portion extracted out.

iii. sets \(D_i = V_i - I[i]\)

iv. If \(D_i \geq 0\), outputs \(P[i] + D_i\), the index of the source symbol.

v. By one-to-one communication operation specified in Lemma-1, the processor whose \(D_i \geq 0\) sends its processor-index \(i\) to the first processor. (There could be only one such processor that turns \(D_i\) to be non-negative)

(2.2) The processor \(P_1\) updates \(base\) to \(base + i\) and broadcasts the updated value to all other processors.
Theorem 5.4.1: Given an input code, the Huffman decoding algorithm for determining one source symbol can be performed in $O(1)$ time on LARPBS model with $O(d)$ processors.

Proof: In our algorithm, the given input code $H$ is converted into a decimal number and it is stored in the first processor. Each array element $P[i]$ from prefix-skip table is stored in a processor $P_i$. The global variable $base$ is stored in the first processor and the processor broadcasts the update values of variable $base$ to the other processors as and when needed. Hence, we avoid the need for shared memory. The time requirements are as follows:

- In the first bus cycle, $P_1$ broadcasts $base$ and $H$ to all the processors which takes $O(1)$ time.

- In the second bus cycle, each processors find the value of $D_i$, and if its non-negative it sends its processor-index $i$ to the first processor using one-to-one communication operation which takes $O(1)$ time as per lemma-1.

As we get the output as indices, it is easy to find the source symbol corresponding to $H$ using these indices. Hence, our algorithm takes $O(1)$ time on an LARPBS model with $O(d)$ processors.

5.5. Time complexity analysis

In our algorithm we invoked two operations broadcasting and one to one communication which are done in $O(1)$ bus cycles on LARPBS model. Though the algorithm has a loop in step 2, it will work fixed number in iteration, radix
being restricted to a constant. Hence we claim our algorithm works in $O(1)$ time with $d$ processors.

5.6. An Illustration

Consider an input code $H = 01111011$. Figure-5.1 represents a single-side growing Huffman tree and the corresponding symbol table, skip table, internal table and the prefix-skip table are shown in the Tables 5.1, 5.2, 5.3 and 5.4 respectively.

In step1, PE1 (processor 1) initialises the base value as 1 and broadcasts base and the decimal form of $H$ to all processors.

In step2, PE3 starts to work which has a Val(011), therefore $V_3 = 3$. Hence the value of $D_3 = 3 - 2 = 1$. Since $D_3 \geq 0$, PE3 outputs the value $2 + 1 = 3$, by taking $P[3] = 2$ from prefix-skip table. The base value is updated as base + 3 = 4.

Now the PE2 has the value Val(11) = 3. So, the value of $D_2 \geq 0$, which gives the output $1 + 0 = 1$, by taking $p[2]$ value as 0 from prefix-skip table. The base value has been again changed to base + 2 = 6. Continuing this way PE4 has value Val(0011) = 3 and gives output $4 + 3 = 7$, since $D_4 \geq 0$. Therefore, we get indices $\{3, 1, 7\}$ and the corresponding symbol from the symbol table is $\{s_3, s_1, s_5\}$.

5.7. Discussion

It is pertinent to point out here that the development of exact or more efficient parallel algorithm for Huffman decoding is deemed of importance from the view of wide practical application of Huffman decoding. Keeping this in
view, we have presented a constant time Huffman decoding parallel algorithm on a LARPBS model and it works without any degradation in running time in comparison with its CREW PRAM counterpart developed in [12]. In fact, many new results achieved so far [45, 47, 50] indicate that this is true especially for many communication intensive problems.

Hence, we have shed some lights on one communication intensive problem in the area of Cellular Manufacturing System (CMS), which is a formation of Machine Cells. Based on Maximal Spanning Tree (MST) approach, we have designed an $O(\log n)$-time parallel algorithm using LARPBS model. This is the focus of our investigation in the next chapter.