A Two-phase Practical Parallel Algorithm for Construction of Huffman Codes

S. Arash Ostadzadeh[†]**, B. Maryam Elahi**[‡]**, Zeinab Zeinalpour Tabrizi**[*]**, M. Amir Moulavi**[*]**, Koen Bertels**[†]

[†] Computer Engineering Laboratory, EEMCS, Delft University of Technology, Delft, The Netherlands
[‡] Center for Parallel Computers, Royal Institute of Technology, Stockholm, Sweden
[*] Computer Engineering Department, Islamic Azad University of Mashhad, Mashhad, Iran

Abstract - *The construction of optimal prefix codes plays a significant and influential role in applications concerning information processing and communication. For decades, different algorithms were proposed treating the issue of Huffman codes construction and various optimizations were introduced. In this paper we propose a detailed practical time-efficient parallel algorithm for generating Huffman codes on CREW PRAM model exploiting n processors, where n is equal to the number of symbols in alphabet. We first compute the codewords lengths for all symbols concurrently with an innovative direct parallelization of the Huffman tree construction algorithm, alleviating the complexity of dealing with the original tree-like data structure. Then Huffman codes corresponding to symbols are generated in parallel based on a recursive formula introduced in [5]. The proposed algorithm achieves an O(n) time in the worst case when one-sided Huffman tree is formed, which is rarely encountered in practice, and O(log((logn – 1)!)) time in the best case when Huffman tree is nearly balanced.*

Keywords: Data Structures, Parallel Algorithms, Optimal Prefix Codes, Huffman Codes, PRAM.

1 Introduction

The construction of optimal codes for a given alphabet is a classical problem with significant and influential applications in information processing and communication. Let $\sum = \{S_1, S_2, \dots, S_n\}$ be an alphabet. A set of codes $\check{C} = \{C_1, C_2, \dots, C_n\}$ over $\sum$ is a finite set of distinct sequences over $\sum$. Each sequence C_i is called *codeword*. A code $\check{C}$ is called a prefix code if no codeword in $\check{C}$ is a prefix of another one in the set. We define a message M over $\check{C}$ to be a concatenation of codewords from $\check{C}$. If the frequency (or probability provided that real value of the frequency can not be determined) of appearance of S_i in M is $P_i \in R$, then the Huffman coding problem is to construct a prefix code $\check{C} = \{C_1, C_2, \dots, C_n \in \sum^*\}$ such that $\sum_{i=1}^{n} P_i * |C_i|$ is minimum among all the possibilities of $\check{C}$, where $|C_i|$ is the length of C_i. As an obvious result, if we assume M is to be transmitted over a communication channel which can transfer one symbol of the alphabet $\sum^*$ per unit of time then the transmission time would be the least possible. It is easy to discover a desirable property in prefix codes that a message can be decomposed in only one way.

In 1952, Huffman [8] proposed an elegant sequential algorithm which generates optimal prefix codes in O($n$log$n$) time. The algorithm actually needs only linear time provided that the frequencies of appearances are sorted in advance [17, 21]. Since then there have been extensive researches on analysis, implementation issues and improvements of the Huffman coding theory in a variety of applications [2, 3, 4, 5, 6, 7, 9, 10, 13, 15, 17, 19, 21]. Researches to address the problem in parallel environments also emerged [1, 11, 14, 16, 18, 20]. There are already several attempts to construct the Huffman codes in parallel. Teng [20] proposed the first NC algorithm to generate Huffman codes using n^6 processors in O(log$n$) time which seems rather unpractical due to the huge number of processors. Atallah et al. [1] showed how to reduce the number of processors to n^3 while maintaining the same time complexity. They also presented an O($\log^2 n$) time, $n^2/\log n$ processors as well as an O(log$n$) time, $n^3/\log n$ processors CREW deterministic parallel algorithms for construction of Huffman codes. Further they concluded that the time can be reduced to O($\log n(\log\log n)^2$) on a CRCW model using only $n^2/(\log\log n)^2$ processors.

Kirkpatrick and Przytycka [11] presented several approximated parallel algorithms for construction of Huffman codes. Later Larmore and Przytycka [16] proposed an O($\sqrt{n}$ log$n$) time algorithm that uses O(n) processors. The original algorithm is developed as a solution for the concave least weight subsequence problem but is extended for the Huffman coding problem. Milidiú et al. [18] proposed a work efficient parallel algorithm on CREW to address the problem. Their algorithm runs in O($H\log\log(n/H)$) time with n processors, where H is the length of the longest Huffman code. Since H is in the interval $[\lceil \log n \rceil, n-1]$, the algorithms requires O(n) time in the worst case. It is known that the length of the Huffman code is bounded to $\log x^{-1}$ where $x = P_1/(\sum_{i=1}^{n} P_i)$ [2]. As a result the time complexity of the algorithm can be considered $O(\log x^{-1} \log \log n)$. The major problem with Milidiú et al.'s algorithm and some other parallel Huffman construction solutions is that instead of actually generating the Huffman codes they rather construct the Huffman tree in parallel and it is not clear how the codes should be built from the Huffman tree concurrently if possible.

In this paper we particularly address this problem by proposing a practical detailed CREW algorithm. In other words the output of our algorithm is the Huffman codes not the Huffman tree. The algorithm is based on a sophisticated parallelization of the direct Huffman tree constructing simulation with the elimination of the need to store nodes in a tree-like data structure. We first compute the path length for each symbol in the Huffman tree and then focus on generating the Huffman codes in parallel by exploiting Hashemian's recursive formula based on single-side growing Huffman tree [5]. Our Algorithm can be implemented in O*(n)* time on the CREW PRAM model incorporating *n* processors in the worst case and the best case time is bounded to $O(\log((\log n - 1)!))$, where *n* equals the number of symbols.

The rest of this paper is organized as follows. In Section 2, we describe the fundamental structures used in our pseudocode and the definitions. In section 3, we first give an outline of our algorithm and then it is examined in details. The performance analysis is discussed in section 4. We conclude in section 5.

2 Data structures

To generate the Huffman codes for a given set of symbols, we need to know the position of each symbol in a tree known as Huffman tree. The codeword for each symbol could be obtained by traversing the tree from the root to the leaf associated with that symbol; along the path every left branch counts as a '0' and every right branch counts as a '1'. Symbols can only be the leaves of the Huffman tree; therefore the code generated from this tree is a prefix code and hence, has a unique decomposition.

We assume that the input to the first phase of our algorithm is a symbol table including an array of symbols $S = \{s_1, s_2, \ldots, s_n\}$ and an array of corresponding frequencies $F = \{f_1, f_2, \ldots, f_n\}$. This symbol table is sorted based on frequencies in non-decreasing order. Each symbol corresponds to a leaf in the Huffman tree. We define a structure for these leaves, including *freq*, a field for frequency value, and *leader*, a pointer to the root of the subtree that this leaf belongs to. *lNodes,* an array of the mentioned structure, shows the leaders of the leaves that have already participated in the construction of the tree levels, in which $lNodes_i$ corresponds to s_i. We define a similar structure for internal nodes. *iNodes* behaves as a queue of this data structure. As new tree levels are generated, new internal nodes are added to the queue. Array *Temp* is a data structure for temporary storage of a merged list of internal nodes from *iNodes* and leaf nodes from *lNodes,* which are the nodes participating in the construction of each new tree level. These nodes are then melded to form the internal nodes of the next higher level. Each element of *Temp* contains three fields: *freq*, *isLeaf* and *index*. The chosen leaf nodes, who participate in construction of the current level, are copied to *Copy,* which is an array with the same structure as *Temp*. The first phase generates an array of codeword lengths $CL = \{cl_1, cl_2, \ldots, cl_n\}$ in which cl_i is the codeword length for s_i, such that $cl_i \leq cl_{i+1}$. The second phase of our algorithm takes the array of codeword lengths *CL* as input and generates the final codewords in $CW = \{cw_1, cw_2, \ldots, cw_n\}$, in which cw_i is the codeword for s_i.

3 Algorithm

In this section, we first portray the outline of our algorithm and then describe the details of implementation issues concerning each phase.

3.1 Outline

We propose a two-phase parallel algorithm for time-efficient construction of Huffman codes on the CREW PRAM model. Our algorithm requires *n* processors, which is equal to the number of input symbols. The two phases of the presented algorithm are: Codeword Length Generation (*CLGeneration*) and Codeword Generation (*CWGeneration*). The outline is depicted in Fig. 1.

In the first phase, *CLGeneration* algorithm computes the codeword length for each symbol s_i which is equal to the path length of s_i in the Huffman tree. This is done without maintaining a tree structure in practice and by generating the tree levels, one at a time, in a bottom-up fashion. With generation of each new level in the tree, the codeword lengths of all symbols whose leaders have participated in the construction of the new level are incremented. We define a symbol's leader as the root of the subtree that the symbol belongs to. At each level, the two nodes with smallest frequencies are found among the internal nodes constructed in the previous iteration and leaf nodes who have not participated in the construction of the tree yet. These two are combined to form a new internal node. The combined value indicates the minimum frequency of the next higher level, which plays a crucial role in selecting leaf nodes for participation in the current level. At this point, all the internal nodes that belong to the previous level and leaf nodes who have not participated yet and have a frequency smaller than or equal to the minimum frequency, are selected, merged and melded pair-wise to form new internal nodes. The new internal nodes are the new leaders in their subtrees; hence leaf nodes need to check whether or not their leaders have changed. In case of a change, they update their leaders and increment their codeword lengths. This process is repeated until only one internal node remains, which is the root of the tree. At the end of the first phase, *CL* has the codeword lengths for all

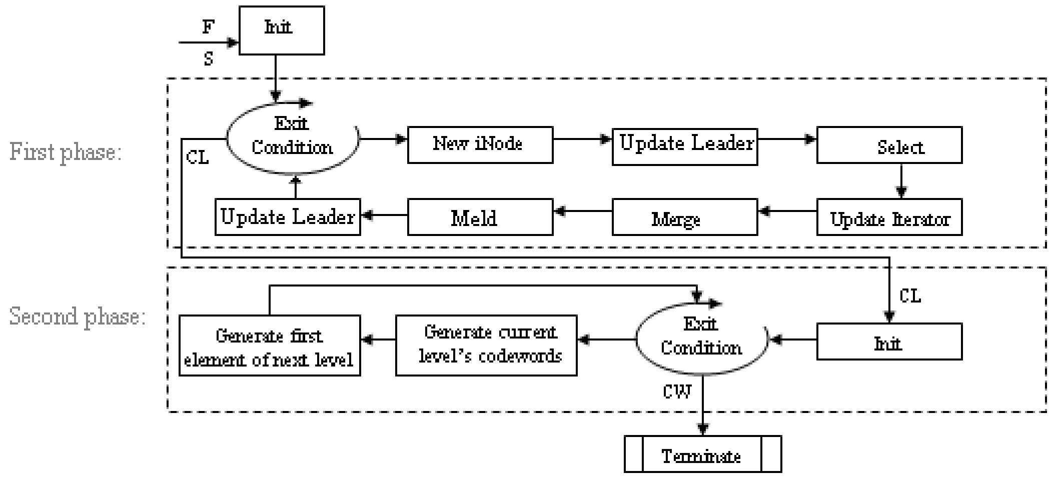


Fig. 1. Algorithm Outline

symbols in non-increasing order.

Codeword generation is performed in a top-down fashion; therefore we need to reverse *CL* in the initialization stage of the second phase. The final Huffman code for each symbol is generated from the codeword length with the help of a parallel version of the recursive formula introduced in [5]. In proposition 2, we show that the codewords for symbols with the same codeword lengths can be constructed in parallel. For each group of symbols with the same codeword lengths, which are symbols residing in the same level of the tree, the codeword of the first symbol is computed and then the rest of the group generate their codewords in parallel. This process is repeated for each level of the tree until *CW* has the final codewords for all symbols.

Proposition 1. Let the trees $T_1, T_2, ..., T_k$ with the corresponding frequencies $fr_1, fr_2, ..., fr_k$ such that $fr_i \leq fr_{i+1}, 0 < i < k$ be present in a forest at stage *s* of the Huffman Tree construction algorithm. For all trees $T_3, ..., T_m$ such that $fr_i \leq fr_1 + fr_2$, $3 \leq i \leq m$, the trees T_{2j+1} and T_{2j+2}, $1 \leq j \leq \lfloor (m-2)/2 \rfloor$ can be melded in parallel.

Proof. Since T_1 and T_2 hold the minimum frequencies among all the trees, the combination of T_1 and T_2 is accomplished by the definition of Huffman tree construction algorithm.

If we assume that the combinations of T_3 and T_4, T_5 and T_6 and so on, are not performed then two alternative assumptions should be investigated. First, suppose two trees T_s and T_{s+1} (two adjacent trees according to their frequencies) residing in the current forest, are not combined in subsequent stages of the Huffman tree construction algorithm, instead T_s is combined with $T_{s\pm\varepsilon}$, $\varepsilon \geq 2$. This assumption is rejected due to the definition of the Huffman tree construction algorithm, indicating two trees with minimum frequencies are selected at each stage, and there exists at least one tree with the frequency lower than $T_{s+\varepsilon}$ which can be selected for the combination. Similarly, the combination of T_s and $T_{s-\varepsilon}$ can not be true, because $fr_{s-1} \leq fr_s$ and if a combination should be performed T_s would not be a valid candidate.

Second, we can make an assumption that the combination of T_s, $3 \leq s \leq m$ is carried out with another tree T_p whose creation is conditioned on passing the current stage of the algorithm, i.e. the tree is not created yet and it doesn't exist in the forest. This assumption is also rejected because $fr_p \geq fr_1 + fr_2$, which means the to-be-created trees in subsequent stages have a lower bound of (fr_1+fr_2) for their frequencies. However, if there exists candidate trees in the forest presently, their frequencies are limited to (fr_1+fr_2) at most which can be considered for the combination process in the current stage and there is no need to wait for the next stage to come.

Since none of the alternative assumptions are true, the proof is complete by contradiction. ■

Proposition 2. Codeword generation can be performed in parallel for those symbols with the same codeword length.

Proof. The proof is trivial. In the recursive codeword generation formula proposed by Hashemian [5], the value of the codeword for symbol *s* is dependant on the codeword value of its precedent symbol, however for all those symbols with the same codeword length, the value of $CL_{i+1} - CL_i$ equals zero, as a result the formula is revised to $C_{i+1} = C_i + 1$ which means that if we have the codeword value of the first symbol in the sequence of equal-codeword-length symbols, for all the subsequent symbols following the first one, the codeword value of a symbol in the sequence is one greater than the previous one. Provided that we have *k* processors, each responsible for a symbol in the sequence, knowing the codeword value of the first symbol, the codeword generation for these *k* symbols can be performed in parallel. Each processor only needs to know the distance of its symbol from the first one in the sequence and add this value to the first symbol's codeword. Hence, we have the proposition. ■

3.2 Description

The following two subsections discuss each phase of the algorithm in details and they are accompanied by an example for clarification.

3.2.1 CLGeneration

For the *CLGeneration* phase, first we need to initialize the basic structures as depicted in Fig. 2.

The following arrays are initialized in parallel. *lNodes* is an array corresponding to *S* containing a frequency field which is initialized with the frequency of its symbol and its leader is set to -1. *CL*, the array that shows the codeword length for each symbol, is initialized with 0. Next, processor P_1 sets the following variables. *lNodesCur* points to the last leaf node that has participated in the construction of a level so far and it is initialized with 0. *iNodesFront* and *iNodesRear* are the front and rear indicators for *iNodes*, which behaves as a queue and shows the newly generated internal nodes who have not participated in the construction of a level. They are initializes with zero, indicating that the *iNodes* queue is empty.

```
Forall processors P_i (1 ≤ i ≤ n) do in parallel
    lNodes[i].freq ← F[i]
    lNodes[i].leader ← -1
    CL[i] ← 0
P_1 sets
    iNodesFront ← 0
    iNodesRear ← 0
    lNodesCur ← 0
```

Fig. 2. Initialization

After initialization, the following operations are performed iteratively until all leaves are processed and no internal node is left in the *iNodes* queue, except the root. The sum of two minimal frequency values, *MinFreq*, determines the frequency value of the next internal node. This internal node could be constructed from the combination of two new leaves, an internal node and a new leaf or two internal nodes. In case of ties, leaves are preferred to participate in the construction of the new internal node. Here, array *mid* and *SelectMinimums* function are used to make the code concise and simplify the process. The new internal node is added to the *iNodes* queue as illustrated in Fig. 3. The leaders of the two internal nodes or leaves who have been combined are set to the index of the new generated internal node in the *iNodes* queue.

The select module takes *lNodes* and *MinFreq* as input parameters returning *Copy* and *CurLeavesNum* as output parameters. It copies those leaves whose indexes are more than *lNodesCur* and their *freq* is less than or equal to *MinFreq*, to the *Copy* array. *Copy* array has three fields: *freq*, *isLeaf* and *index*. *freq* is the value of the selected leaf frequency; *isLeaf* in this step has the value of true for all elements because they are all leaves and *index* is the index of the selected leaves in *lNodes*. *CurLeavesNum* is the number of selected leaves that have been copied to the *Copy* array. Fig. 4 illustrates the Select module.

```
P_1 sets
    mid ← {∞, ∞, ∞, ∞}
    if (lNodesCur ≤ n – 1)
        mid [1] ← lNodes[lNodesCur+1].Freq
    if (lNodesCur ≤ n – 2)
        mid [2] ← lNodes[lNodesCur+2].Freq
    if (iNodesRear > iNodesFront)
        mid [3] ← iNodes[iNodesFront+1].Freq
    if (iNodesRear > iNodesFront + 1)
        mid [4] ← iNodes[iNodesFront+2].Freq
    SelectMinimums (mid)
    MinFreq ← mid[1] + mid[2]
    iNodes [iNodesRear + 1].freq ← MinFreq
    iNodes [iNodesRear + 1].leader ← -1
    if (isLeaf (mid[1]))
        lNodes[lNodesCur+1].leader ← iNodesRear + 1
        CL[lNodesCur+1]++, lNodesCur++
    else
        iNodes[iNodesFront + 1].leader ← iNodesRear + 1
        iNodesFront++
    if (isLeaf(mid[2]))
        lNodes[lNodesCur+1].leader ← iNodesRear + 1
        CL[lNodesCur+1]++,  lNodesCur++
    else
        iNodes[iNodesFront + 1].leader ← iNodesRear + 1
```

Fig. 3. New iNode

The module that is illustrated in Fig. 5 is used for determining the participating elements in the construction of the current level. The values of *CurLeavesNum*, *MergeFront*, and *MergeRear* indicate which nodes in the *Copy* and *iNodes* arrays are merged. If the total number of unprocessed *iNodes* elements and *CurLeavesNum* is odd, the module excludes an internal node or leaf node that has the maximum frequency; in case of ties, it is preferred to keep leaf nodes. In case the *iNodes* queue is empty, the *CurLeavesNum* is decremented, thus the length of *Temp* array becomes even. *MergeFront* and *MergeRear* denote the beginning and the end of the segment in *iNodes* that participates in the Merge function.

```
Forall processors P_i (lNodesCur < i ≤ n)
    if (lNodes[i].freq ≤ MinFreq)
        Copy[i – lNodesCur].freq ← lNodes[i].freq
        Copy[i – lNodesCur].index ← i
        Copy[i – lNodesCur].isLeaf ← true
        if (i = n || lNodes[i+1].freq > MinFreq)
            CurLeavesNum ← i – lNodesCur
```

Fig. 4. Select Module

The Merge function performs the task of combining two sorted lists in O(loglog*n*) on CREW PRAM model [12]. The function accepts *Copy, CurLeavesNum, mergFront* and *mergRear* as input parameters and *Temp* and *TempLength* as output. The main task of this function is to build the *Temp* array which is the combination of *Copy* array and the segment in *iNodes* that is indicated by *MergeFront* and *MergeRear*. *Temp* is sorted based on the *freq* field in non-decreasing order.

```
P1 Sets
  mergeRear ← iNodesRear
  mergeFront ← iNodesFront

  if((CurLeavesNum+ iNodesRear - iNodesFront)%2=0)
    iNodesFront ← iNodesRear

  else if ((iNodesRear - iNodesFront != 0) &&
  (F[lNodesCur+CurLeavesNum]≤iNodes[iNodesRear].freq))
    mergeRear--
    iNodesFront ← iNodesRear - 1
  else
    iNodesFront ← iNodesRear
    CurLeavesNum --

  lNodesCur ← lNodesCur + CurLeavesNum iNodesRear++
```

Fig. 5. Updating Iterators

In the next step, the Meld module generates the new internal nodes of the next level. Each processor is assigned to two consecutive elements of *Temp* according to its index. These pairs of elements are melded to form new internal nodes whose *freq* is the sum of the combined pairs' frequencies. Then the corresponding processor updates the *leader* fields of the two participating nodes. The location of each node is determined by the *isLeaf* field indicating that the element resides in *lNodes* or *iNodes* array.

```
Forall processors Pi (1 ≤ i ≤ TempLength) do in parallel
    ind ← iNodesRear + i
    iNodes [ind].freq ← temp [2*i-1].freq + temp [2*i].freq
    iNodes[ind].leader ← -1

    if (temp [2*i-1].isleaf)
      lNodes [temp [2*i – 1].index].leader ← ind
      CL[temp [2*i – 1].index]++
    else
      iNodes [temp [2*i – 1].index].leader ← ind
    if (temp [2*i].isleaf)
      lNodes [temp [2*i].index].leader ← ind
      CL[temp [2*i ].index]++
    else
      iNodes [temp [2*i].index].leader ← ind
P1 sets
      iNodesRear ← iNodesRear + (TempLength/2)
```

Fig. 6. Meld Module

In the end, P_1 increments the *iNodeRear* based on the number of the newly added internal nodes which is equal to half of *TempLength*. *lNodesCur* is incremented by the value of *CurLeavesNum* which is equal to the number of leaves who participated at this level.

If the leader of an internal node is changed, all its children need to update their leader and set it to the index of the new internal node. The codeword lengths corresponding to these leaf nodes are also incremented. This is done in parallel. In this step all leaves check their leaders in parallel and they figure out whether or not their leaders have changed. If their leaders are assigned to new leaders, they update their leaders by getting the value of their leader's leader. This is depicted in Fig. 7.

```
Forall processors Pi (1 ≤ i ≤ n) do in parallel
  if (lNodes[i].leader != -1)
    if (iNodes[lNodes[i].leader].leader != -1)
      lNodes[i].leader ← iNodes[lNodes[i].leader].leader
      CL[i] ++
```

Fig. 7. Update Leaders

The mentioned process repeats until only one internal node remains, which is the root. At this point, *CL* has the codeword lengths for all symbols.

The example in Fig. 8 depicts the progress in the first pass of *CLGeneration* for a given input. The process is repeated for every level in the tree and in the end, *CL* has the codeword lengths for all symbols in *S*.

3.2.2 CWGeneration

In the *CWGeneration* phase, we generate the final codewords by introducing a parallel version of the algorithm proposed by Hashemian [5]. *CL* is the input array for this phase with the length of *n* and *CW* is the output array with the same number of elements. We generate the final codewords in a top-down fashion, so we need to reverse *CL*. Reversing is a simple process that is accomplished in parallel. In the pseudocode depicted in Fig. 10, the variables *CCL* and *CDPI* are used to show the current codeword length and current done-processor index respectively. In each iteration, *CCL* (Current Codeword Length) has the length of the codewords of the current level and *CDPI* (Current Done-Processor Index) indicates the index of the last processor who has finished generating its codeword. Generation of codewords for each level is accomplished in two steps, codeword of the first symbol in the level is computed and then all other symbols with the same codeword lengths construct their codewords. While we have the codeword lengths of all symbols, we can compute the codewords by employing the following recursive formula [5]:

$$C_{i+1} = (C_i + 1) * 2^{cl_{i+1} - cl_i} \qquad (1)$$

Initialization is done by the first processor. The value of *CCL* is set to the value of the first element of *CL*. A string of zeros with length of *CCL* is put in the first cell of the array *CW*. For the processors whose corresponding symbol's codeword length is equal to *CCL*, the construction of the codewords can be accomplished in parallel because they only have to add a number to the codeword of the first symbol in the series. P_1 generates the first codeword of the next group to provide processors assigned to the next level with the base value upon which they construct their codewords in parallel.

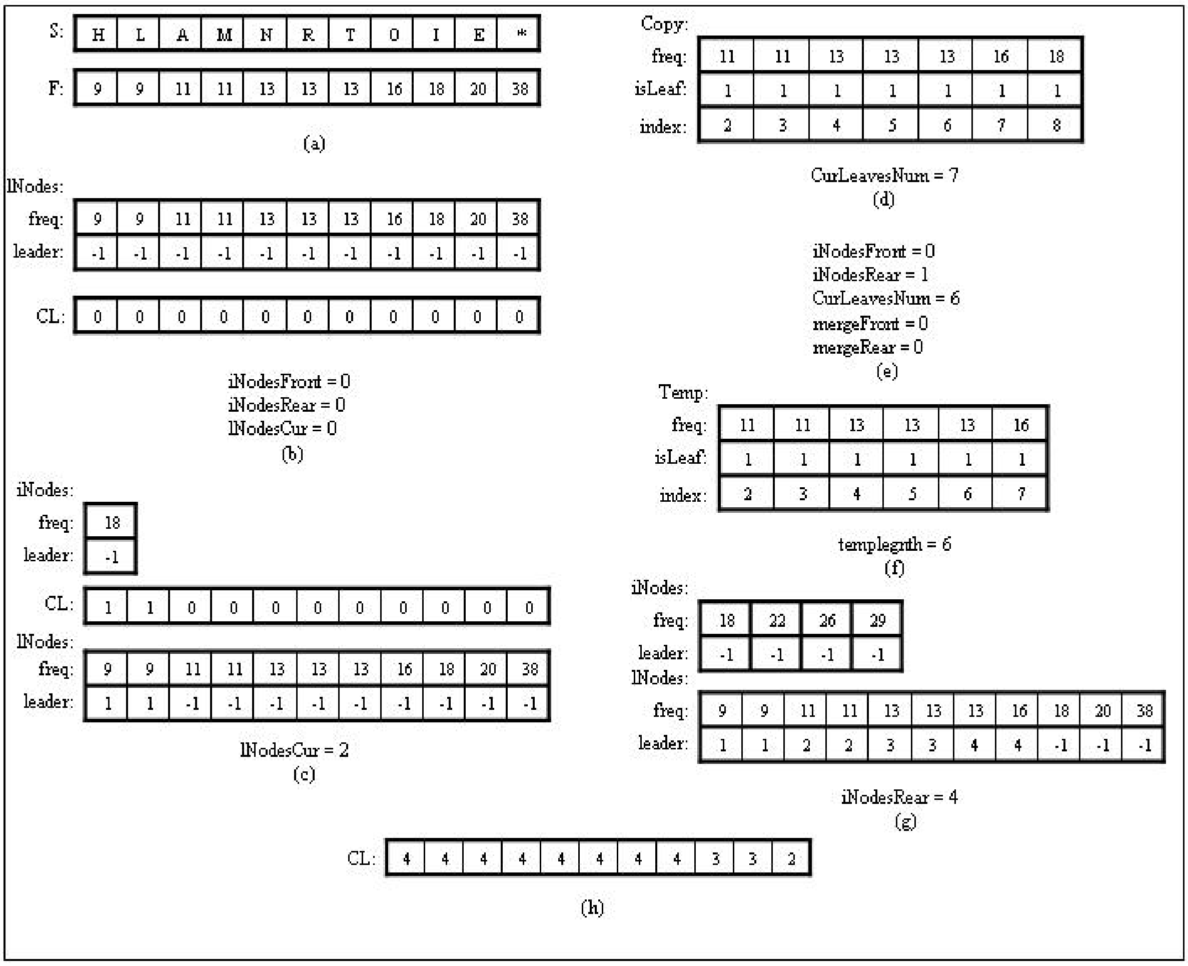
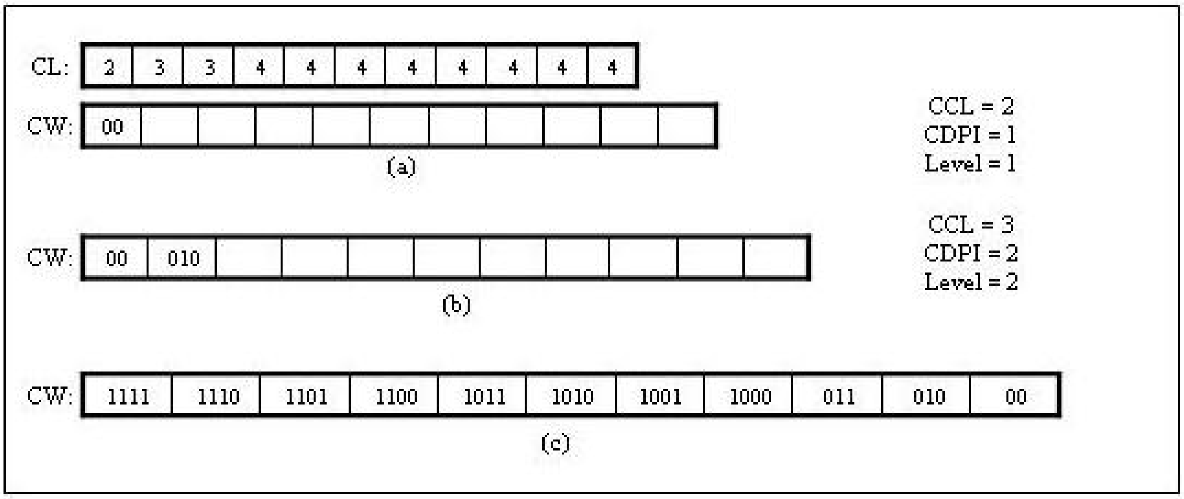


Fig. 8. a) Input to the first phase b) *lNodes* and *CL* initialized before the first cycle c) First internal node is generated d) Select chooses the participating leaf nodes e) Iterators updated to determine the participating *lNodes* and *iNodes* f) *Temp* is filled with a merged list of participating *lNodes* and *iNodes* g) Nodes in *Temp* melded to form new *iNodes*. The leaders are updated and *CL* elements corresponding to participating leaf nodes are incremented h) Final result in *CL* for given input



CL: 2 3 3 4 4 4 4 4 4 4 4
CW: 00
(a)
CCL = 2
CDPI = 1
Level = 1
CW: 00 010
(b)
CCL = 3
CDPI = 2
Level = 2
CW: 1111 1110 1101 1100 1011 1010 1001 1000 011 010 00
(c)

Fig. 9. a) Initialization b) Generation of the first value in the series for the next level c) The output of this phase (codewords)

```
Forall processors P_i (1 ≤ i ≤ n/2)
    LocalVariable ← CL [i]
    CL [n – i + 1] ← LocalVariable

P_1 sets
   CCL ← CL [1]
   CW [1] ← bit string of CCL zeros
   CDPI ← 1

While (CDPI < n)
    Forall processors P_i (1 ≤ i ≤ n) do in parallel
        if (i > CDPI  && CL[i] = CCL)
    CW[i] ← CW [CDPI] + (i - CDPI)
            if (i < n  && CL[i + 1] != CCL)
       CDPI ← i
       else be idle

    P_1 sets
       CLDiff ← CL [CDPI + 1] – CL [CDPI]
       CW [CDPI + 1] ← (CW [CDPI] + 1) * 2^ (CLDiff)
       CCL ← CL [CDPI+1]
       CDPI ← CDPI + 1

Forall processors P_i (1 ≤ i ≤ n/2)
    LocalVariable ← CW[i]
    CW [n – i + 1] ← LocalVariable
```

Fig. 10. Codeword Generation

These iterations continue until the codewords for all symbols are constructed. In the end, *CW* is reversed to make codewords correspond to the right symbols.

The example in Fig. 9 depicts the progress in the *CWGeneration* phase for a given input. The process is repeated for every level in the tree and in the end, *CW* has the codewords for all the symbols in *S*.

4 Performance analysis

First, we analyze each phase of the algorithm separately and then we discuss the performance of the algorithm as a whole.

Lemma 1. *CLGeneration performs L cycles.*

Proof. It can be proved by induction that if *lN* is the first chosen leaf node at the first iteration, that is the leftmost leaf at the bottommost level of the Huffman tree , a new ancestor for *lN* is generated in each cycle [18]. *lN* has (*L*-1) ancestors, hence the number of cycles is equal to *L*.■

Theorem 1. *The CLGeneration runs in O(Lloglog(n/L)) time.*

Proof. *CLGeneration* is comprised of a set of operations which are performed within a number of cycles, one cycle per each level of the tree. Lemma 1 states that the number of these cycles is L, the height of the Huffman tree.

Before the main loop, *Initialization* is performed which has a parallel section and a sequential section, both of O(1) time. Next, at every cycle, a number of operations are performed. *New iNodes* module operates a few sequential comparisons on four nodes in O(1). *Update Iterators* is also of constant order. *Update Leaders* and *Select* and *Meld,* which all execute in parallel, are of O(1) parallel time. This is because a constant number of operations are performed on *i* variables by *i* processors in parallel; such that $1 \le i \le n$. The only part that is not of constant order is the *Merge* operation that can be performed in O(loglog*M(i)*) parallel time [12], in which *M(i)* is the number of internal nodes generated at cycle *i*. So, the time required by *CLGeneration* is as follows.

$$T_1 = O\left(\sum_{i=1}^{L} \log\log M(i)\right) \qquad (2)$$

Since the number of internal nodes in a Huffman tree with *n* leaf nodes is equal to (*n - 1*), M*(i)* is constrained to $\sum_{i=1}^{L} M(i) = (n-1)$. It can be seen that the upper bound is directly dependent on *L*, the height of the tree that is in the interval $[\lceil \log n \rceil, n-1]$. If *L* equals *n*-1, which means we have a one sided tree, M*(i)* is of O(1), hence the algorithm runs in O*(n)*. If the tree is balanced, we have:

$$\sum_{i=1}^{L} \log\log M(i) = \log\log 2 + \log\log 2^2 + \log\log 2^3 + \ldots + \log\log 2^{(\log n)-1} \qquad (3)$$

$$\sum_{i=1}^{L} \log\log M(i) = (0 + \log 2 + \log 3 + \ldots + \log(\log n - 1)) \qquad (4)$$

$$\sum_{i=1}^{L} \log\log M(i) = \log(1 * 2 * 3 * \ldots * (\log n - 1)) \qquad (5)$$

$$\sum_{i=1}^{L} \log\log M(i) = \log(\log n - 1)! \qquad (6)$$

Thus the time complexity for the best case where the tree is balanced is O (log(log*n*-1)!).

In general, we can find an upper bound, by maximizing T_1. Through Jenson's inequality we know:

$$\frac{\sum f(x_i)}{n} \le f\left(\frac{\sum x_i}{n}\right) \qquad (7)$$

Hence, O(*L*loglog(*n/L*)) is an upper bound for T_1. ■

Theorem 2. *The CWGeneration runs in O(L) time.*

Proof. *CWGeneration* has an initialization preprocessing for inverting *CL* and setting the variables, and a post processing phase for inverting *CW* which are both of constant order. The inversion process is performed in parallel in O(1).

The main operations of *CWGeneration* are performed within a loop that cycles until the codewords for all symbols are generated. The operations within the loop consist of a sequential section performing a constant number of assignments and a parallel section performing a few comparisons and assignments, hence both are of O(1). At *i*-th cycle, the codewords for all symbols with the codeword lengths equal to CCL_i are constructed, thus there is one cycle per each level of the tree in which symbols have the same codeword lengths. Therefore *CWGeneration* runs in O(*L*). Since *L* is in the interval $[\lceil \log n \rceil, n-1]$, the best case is O(log*n*) and the worst case is O(n). ■

Having the time complexity for each of the two phases, it is seen that our algorithm for parallel construction of Huffman codes runs in O(Lloglog(n/L) + L), which is directly dependant to L, the height of the tree. Since L is in the interval $[\lceil \log n \rceil, n-1]$ the algorithm runs in O(n) time in the worst case and runs in O(log(log$n$-1)!) in the best case.

5 Conclusions

The significance of Huffman coding is due to its widespread utilization in information processing and particularly in image and data compression techniques. Our major contributions in this paper are the followings. First, we have presented a new time-efficient practical parallel algorithm for construction of Huffman codes without generating a tree in practice. As the output of the proposed algorithm we have the optimal codewords for all given symbols. The presented algorithm is structured in two separated phases on CREW PRAM model.

Second, the worst case time complexity of the algorithm is O(n) incorporating n processors which rarely occurs in practice. n is equal to the number of symbols in a given alphabet. Our new parallel optimal prefix codes construction algorithm achieves an upper bound of O(log(log$n$-1)!) in the best case which is the same as the best known algorithm addressing this problem, thus time is not sacrificed.

6 References

[1] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller and S-H. Teng. "Constructing trees in parallel", *ACM SIGACT, Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 421-431, June 1989.

[2] M. Buro, "On the maximum length of Huffman codes", Information Processing Letters, Vol. 45, No.5, pp. 219-223, April 1993.

[3] H. C. Chen, Y. L. Wang and Y. F. Lan, "A memory efficient and fast Huffman decoding algorithm", Information Processing Letters, Vol. 69, No. 3, pp. 119-122, February 1999.

[4] T. J. Fexguson and J. H. Rabinowitz, "Self-synchronizing Huffman codes", IEEE Trans. Inform. Theory, Vol. 30, No. 4, pp. 687-693, July 1984.

[5] R. Hashemian, "Memory efficient and high-speed search Huffman coding", IEEE Trans. on Comm., Vol. 43, No. 10, pp. 2576-2581, October 1995.

[6] R. Hashemian, "Direct Huffman coding and decoding using the table of code-lengths", Proc. International Conf. on Inform. Technology: Computers and Communications (ITCC '03), pp. 237-241, April 2003.

[7] S. Ho and P. Law, "Efficient hardware decoding method for modified Huffman code", Electronics Letters, Vol. 27, No. 10, pp. 855-856, May 1991.

[8] D. A. Huffman, "A method for the construction of minimum redundancy codes", Proc. IRE, Vol. 40, No. 9, pp. 1098-1101, September 1952.

[9] S. T. Klein, "Skeleton trees for the efficient decoding of Huffman encoded texts", Kluwer Journal of Inform. Retrieval, Vol. 3, No. 1, pp. 7-23, July 2000.

[10] D. E. Knuth, "Dynamic Huffman coding", Journal of Algorithms, Vol. 6, No. 2, pp. 163-180, June 1985.

[11] D. G. Kirkpatrick and T. M. Przytycka, "Parallel construction of near optimal binary search trees", ACM SIGACT, Proc. 2nd Annual ACM Symp. on Parallel Algorithms and Architectures, pp. 234-243, July 1990.

[12] C. Kruskal, "Searching, merging and sorting in parallel computation", IEEE Trans. Computer, Vol. C-32, No. 10, pp. 942-946, October 1983.

[13] L. L. Larmore, "Height restricted optimal binary trees", SIAM Journal on Computing, Vol. 16, No. 6, pp. 1115-1123, December 1987.

[14] Y. Lin and K. Chung, "A space-efficient Huffman decoding algorithm and its parallelism", Journal of Theoretical Computer Science, Vol. 246, No. 1-2, pp. 227-238, September 2000.

[15] L. L. Larmore and D. S. Hirschberg, "A fast algorithm for optimal length-limited Huffman codes", Journal of ACM, Vol. 37, No. 3, pp. 464-473, July 1999.

[16] L. L. Larmore and T. M. Przytycka, "Constructing Huffman trees in parallel", SIAM Journal on Computing, Vol. 24, No.6, pp. 1163-1169, December 1995.

[17] A. Moffat and J. Katajainen, "In-place calculation of minimum-redundancy codes", 4th Intl. Workshop on Algorithms and Data Structures, Vol. 955, pp. 393-402, August 1995.

[18] R. L. Milidiú, E. S. Laber and A. A. Pessoa, "A work efficient parallel algorithm for constructing Huffman codes", Proc. Data Compression Conference (DCC '99), pp. 277-286, March 1999.

[19] A. Moffat and A. Turpin, "On the implementation of minimum-redundancy prefix codes", IEEE Trans. Commun., Vol. 45, No. 10, pp. 1200-1207, October 1997.

[20] S-H. Teng, "The construction of Huffman-equivalent prefix code in NC", ACM SIGACT News, Vol. 18, No.4, pp. 54-61, 1987.

[21] J. Van Leeuwen, "On the construction of Huffman trees", 3rd International Colloquium on Automata, Languages and Programming, pp. 382-410, July 1976.