A Work Efficient Parallel Algorithm for Constructing Huffman Codes

Ruy Luiz Milidiú
milidiu@inf.puc-rio.br
Eduardo Sany Laber
laber@inf.puc-rio.br
Artur Alves Pessoa
artur@inf.puc-rio.br

PUC-RioInf.MCC16/98 April, 1998

Abstract:

Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ and a corresponding list of weights $[w_1, \ldots, w_n]$, a Huffman code for this alphabet is a prefix code that minimizes the *weighted length* of a code string, defined to be $\sum_{i=1}^{n} w_i l_i$, where l_i is the length of the code assigned to a_i.

In this paper we present ES-ParHuff, a work efficient PRAM CREW algorithm for constructing Huffman codes. This algorithm is a simple and direct parallelization of Huffman's algorithm. The algorithm runs in $O(H \log\log(n/H))$ time with $O(n)$ work for a sorted list of n weights, where H is the length of the longest code generated. For p processors, the algorithm runs in $O(n/p + H \log\log(n/H))$ time. We also prove that the complexity time of ES-ParHuff is $O(\min(n, \log(1/p_1) \log\log n)$, where $p_1 = \sum_{i=1}^{n} w_i$. Under suitable conditions, this algorithm overcomes the best known parallel algorithms for constructing Huffman trees.

Keywords: Prefix Codes, Huffman trees, Parallel Algorithms

Resumo: Dado um alfabeto $\Sigma = \{a_1, \ldots, a_n\}$ com uma lista correspondente de pesos $[w_1, \ldots, w_n]$, um Código de Huffman para este alfabeto é um código de prefixo que minimiza o comprimento médio, definido por $\sum_{i=1}^{n} w_i l_i$, onde l_i é o comprimento do código atríbuido a a_i.

Neste artigo apresentamos o ES-ParHuff, um algorimo ótimo em termos de número de operações para construir códigos de Huffman em uma arquitetura paralela PRAM CREW. Este algoritmo é uma simples e direta paralelização do algoritmo de Huffman. A complexidade de tempo deste algoritmo é $O(H \log\log(n/H))$ e o total de operações é $O(n)$ para uma lista ordenada de n pesos, onde H é o comprimento do maior código gerado. Para p processadores, o tempo de execução deste algoritmo é $O(n/p + H \log\log(n/H))$. Provamos também que sua complxidade de tempo é $O(\min(n, \log(1/p_1) \log\log n)$, onde $p_1 = w_1 / \sum_{i=1}^{n} w_i$. Sob determinadas condições, o ES-ParHuff supera os melhores algoritmos paralelos existentes para construção de códigos de Huffman.

Palavras-Chave: Códigos de Prefixo, Árvores de Huffman, Algoritmos paralelos

1 Introduction

Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ and a corresponding list of weights $[w_1, \ldots, w_n]$, a Huffman code for this alphabet is a prefix code that minimizes the *weighted length* of a code string, defined to be $\sum_{i=1}^{n} w_i l_i$, where l_i is the length of the code assigned to a_i.

A Huffman code can be generated in $O(n \log n)$ time for an unsorted list of weights and in $O(n)$ time if the weights are already sorted [Huf52, vL76, MK95, MPL98]. There are some parallel algorithms for constructing Huffman codes. The best NC algorithm is due to Atallah et all [AKL$^+$89]. It runs in $O(\log^2 n)$ time and requires $O(n^2/\log n)$ processors. In [LP95] Larmore and Prytzcka show a reduction from the CLWS problem to the Huffman code problem and they present an $O(n^{1/2} \log n)$ algorithm for CLWS problem that requires $O(n)$ processors. This last result is improved in [Czu92], where an $O(n^{1/2} \log^{1/2} n)$ algorithm that requires $O(n)$ processors is presented.

A list of n numbers can be sorted in $O(\log n)$ parallel time, with $O(n \log n)$ work [Col88]. Then, throughout this work, we assume that the list of weights is sorted, that is, $w_1 \leq \ldots \leq w_n$.

In this paper, we present ES-ParHuff, a work efficient PRAM CREW algorithm for constructing Huffman codes. An important feature of the algorithm is its simplicity. This algorithm is a direct parallelization of Huffman's algorithm [Huf52, vL76]. It explores a generalization of the key idea explored by the sequential version. ES-ParHuff runs in $O(H \log\log(n/H))$ time with $O(n)$ work, where H is the length of the longest generated code, that is, $H = l_1$. Since H is in the interval $[\lceil \log n \rceil, n-1]$, then in the worst case the algorithm is $O(n)$ and in the best case it is $O(\log n \log\log n)$. We also prove that the complexity time of ES-ParHuff is $O(\min(n, \log(1/p_1) \log\log n)$, where $p_1 = w_1/\sum_{i=1}^{n} w_i$. Hence, our algorithm overcomes Larmore and Przytcka's algorithm [LP95] when $\log(1/p_1)$ is $o(\sqrt{n} \log n/\log\log n)$ and Atallah's [AKL$^+$89] algorithm when $\log(1/p_1)$ is $o(\log^2 n/\log\log n)$.

This paper is organized as follows. In section 2, we present the ParHuff algorithm that runs in $O(H \log\log n)$ time, with $O(nH)$ work. In section 3, we refine ParHuff to obtain the ES-ParHuff that runs in $O(H \log\log(n/H))$ time, with $O(n)$ work. In section 4, we present our final comments.

2 A Parallel Algorithm for Huffman Coding

The problem of finding a Huffman code for a given list W of weights $w_1, \ldots, w_n$ is equivalent to finding a binary tree with minimum weighted external path length $\sum_{i=1}^{n} w_i l_i$ among the set of all binary trees with n leaves. Here, the values l_i for $i = 1, \ldots, n$ are the levels of the leaves in the tree. The equivalence is due to the fact that every prefix code can be represented by a binary tree [Eve79].

Optimal trees are not unique. Huffman's algorithm [Huf52] constructs specific optimal trees, based on a greedy approach. The trees constructed by Huffman's algorithm are usually called Huffman trees.

2.1 Huffman's Algorithm

Huffman's algorithm can be described as a construction of a binary tree, where each weight in W is assigned to a leaf in the tree and the corresponding codeword length is defined as its level in the tree. If s is either a leaf or an internal node, we use $w(s)$ to indicate the weight of node s.

The algorithm starts with a list of leaves S. In the general step, the algorithm selects the two nodes with smallest weights in the current list of nodes S and removes them from the list. Next, the two removed nodes are melded to form a new internal node that is inserted in S. To this internal node is assigned a weight equal to the sum of the weights of its children. The general step repeats until there is only one node in S, the root of the tree.

The correctness of the algorithm is assured by recursively applying the following proposition.

Proposition 1 *Consider a list of nodes* $[s_1, \ldots, s_n]$, *with* $w(s_i) \leq w(s_{i+1})$, *for* $i = 1, \ldots, n-1$. *Then, there is an optimal tree, where* s_1 *and* s_2 *are siblings.*

Proof: It follows immediately from lemma 17.2 in [CLR92]. ■

Figure 1 show the initial steps of a Huffman tree construction for an alphabet with corresponding weights given by $[2, 3, 3, 3, 4, 6, 7, 10, 12]$. In step 1, the leaves with weights 2 and 3 are melded to form an internal node with weight 5. In step 3, an internal node with weight 5 is melded with a leaf with weight 4 to form an internal node with weight 9.

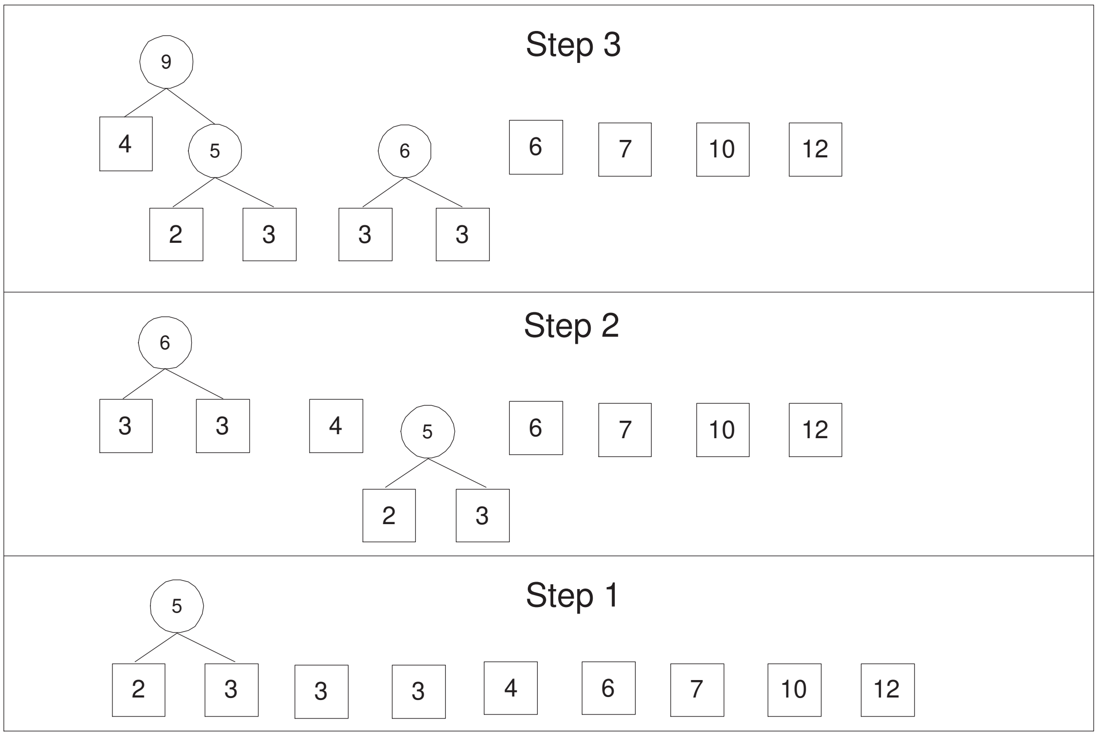

Figure 1: Huffman's algorithm

2.2 The ParHuff Algorithm

In this section we present the ParHuff Algorithm. ParHuff is a direct parallelization of Huffman's algorithm. The basis for the algorithm is the following proposition.

Proposition 2 *Let $s_1, \ldots, s_m$, with $w(s_i) \leq w(s_{i+1})$ be the sorted list of the nodes that haven't been melded during an execution of sequential Huffman's algorithm. If k is the greatest index such that $w(s_k) \leq w(s_1) + w(s_2)$, then the nodes s_{2i-1} and s_{2i}, for $i = 1, \ldots, \lfloor k/2 \rfloor$ are melded by sequential Huffman's algorithm.*

Proof: We can prove this proposition by induction. Our hypothesis is that if $j < \lfloor k/2 \rfloor$, then the nodes s_{2i-1} and s_{2i}, for $i = 1, \ldots, j$, are melded by Huffman's algorithm. If $j = 1$, the result obviously holds. Now, we prove the result for $j + 1$. By hypothesis the nodes s_{2i-1} and s_{2i}, for $i = 1, \ldots, j$ are melded by Huffman's algorithm. Hence, after those meldings, the list S contains the nodes $t_1, \ldots, t_j, s_{2j+1}, s_{2j+2}, \ldots, s_m$, where t_i is the parent of s_{2i-1} and s_{2i}. Since $j + 1 \leq \lfloor k/2 \rfloor$, then

$$w(s_{2j+1}) \leq w(s_{2j+2}) \leq w(s_k) \leq w(s_1) + w(s_2) = w(t_1) \leq w(t_i),$$

for $i = 1, \ldots, j$, and as a consequence, $s_{2(j+1)-1}$ and $s_{2(j+1)}$ are the nodes with smallest weights in S. Therefore, they are melded by Huffman's algorithm.■

This last proposition is in some sense a generalization of proposition 1. Since the nodes s_{2i-1} and s_{2i} for $i = 1, \ldots, \lceil k/2 \rceil$ are melded by Huffman algorithm, then there is an optimal tree where the nodes s_{2i-1} and s_{2i} for $i = 1, \ldots, \lceil k/2 \rceil$ are siblings, what extends proposition 1.

ParHuff Algorithm

```
S ← sorted list of leaves
While length(S) > 1
    Create t_1 as a parent of s_1 and s_2; w(t_1) ← w(s_1) + w(s_2)
    Insert t_1 in T
    Remove s_1 and s_2 from S
    For i = 2 to ⌊length(S)/2⌋ pardo
        If w(s_2i) ≤ w(t_1) then
            Create t_i as a parent of s_2i-1 and s_2i; w(t_i) ← w(s_2i-1) + w(s_2i)
            Insert t_i in T
            Remove s_2i-1 and s_2i from S
        End If
    End For
    S ← Merge (S, T)
    T ← nil
End While
```

Figure 2: A pseudo-code for the ParHuff algorithm.

Proposition 2 allows melding some nodes in parallel. Now, we describe the ParHuff algorithm. The algorithm starts with the list S of leaves $s_1, \ldots, s_n$ sorted by the non-decreasing order of their weights. At the general step, the processor P_1 is responsible for creating the internal node t_1 by melding the nodes s_1 and s_2. The remaining processors check if $w(s_{2i})$ is smaller than $w(t_1)$, for $i = 2, \ldots, \lfloor length(S)/2 \rfloor$. If that happens, the nodes s_{2i-1} and s_{2i} are melded to form a new internal node and are removed from S. Otherwise, they are not melded. After that operation, two sorted lists are obtained. The first list S contains the nodes that were not melded and the second list T contains the internal nodes formed at the current step. These two lists are merged in parallel to form a new sorted list of nodes S, terminating the step. This step repeats until the list S of nodes has only one node, the root of the tree. A pseudo-code for the algorithm is presented in figure 2.

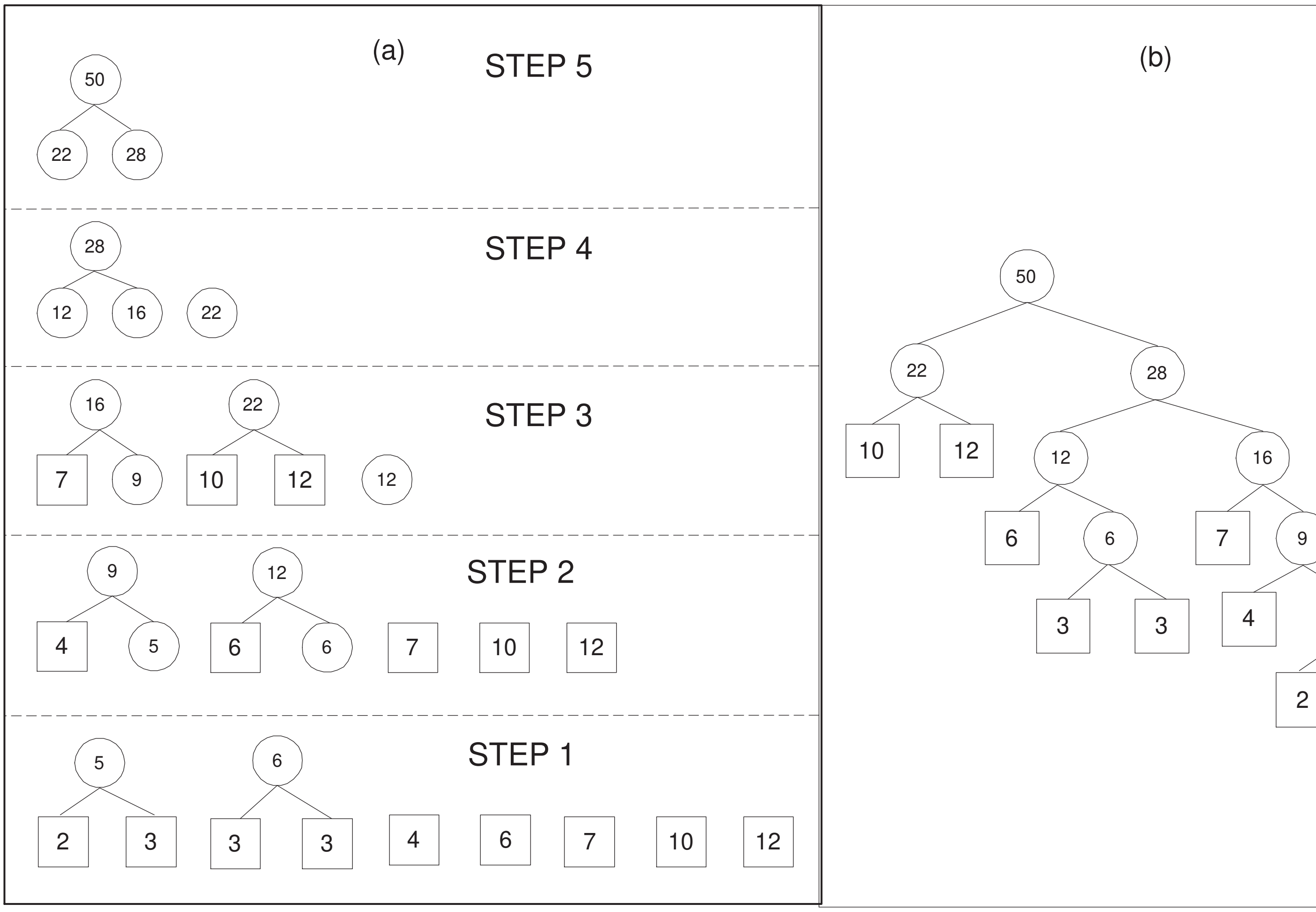

Figure 3: The execution of ParHuff algorithm for the list of weights $[2, 3, 3, 3, 4, 6, 7, 10, 12]$

Figure 3.(a) illustrates the execution of the ParHuff algorithm for constructing a Huffman tree for an alphabet with weights $[2, 3, 3, 3, 4, 6, 7, 10, 12]$. The leaves are represented by squares and the internal nodes by circles. Each step is delimited by dashed lines. As an example, at step 1 two internal nodes are formed, one by melding a leaf with weight 2 and a leaf with weight 3 and another one by melding two leaves with weight 3. The leaves with weights 4 and 6 are not melded since $6 > 3 + 2$. The algorithm performs 5 steps to obtain the final tree presented in figure 3.(b).

Now, we analyze the running time and the work of the algorithm. We use the term cycle to refer to an execution of the external **while** loop.

Lemma 1 *The ParHuff algorithm runs in $O(t \log\log n)$ parallel time, with $O(tn)$ work, where t is the number of executed cycles.*

Proof: At each cycle, the algorithm performs two sets of operations. The first set corresponds to the operations performed inside the **For** command. These operations are responsible for deciding which nodes must be melded to form a new internal node. The second set corresponds to the merge operation.

The number of operations inside the **For** command is $O(n)$ and they can be performed in $O(1)$ parallel time by using $\lfloor n/2 \rfloor$ processors. Furthermore, a parallel merge can be executed in $O(\log\log n)$ time requiring $O(n/\log\log n)$ processors [Kru83], what implies in $O(n)$ work. Since these operations are executed t times, we can conclude that the running time of the algorithm is $O(t \log\log n)$ and its work is $O(tn)$, where t is the number of cycles executed by the algorithm. ■

Let H be the height of the Huffman tree produced by ParHuff. Now, we prove that the number of executed cycles is equal to H.

Lemma 2 *The number t of cycles in an execution of ParHuff is equal to H.*

Proof: Let s_* be the first leaf of S at the beginning of the execution of ParHuff. We prove the correctness of the lemma by showing that a new ancestor of the leaf s_* is formed inside each cycle. Since the number of ancestors of s_* is H, then the number of cycles must be equal to H. This result is enough to establish the lemma.

We prove by induction on the number of cycles, that a new ancestor s_*^k of s_* is formed at each cycle k by melding the two current nodes with smallest weights. For the first cycle the result holds, since s_* is the node with minimum weight at this cycle. Let us assume that after k cycles the result holds. Now, we prove that the result holds after the $(k+1)$-th cycle. Let s_*^k be the ancestor of s_* formed in the k-th cycle. The inductive hypothesis assures that s_*^k was formed by melding the two nodes with smallest weight in the k-th cycle. Hence, all internal nodes formed in the k-th cycle have weights greater than or equal than $w(s_*^k)$. Furthermore, at most one node that was not melded in the k-th cycle has weight smaller than or equal to $w(s_*^k)$, otherwise this node would have been melded in cycle k. Hence, s_*^k is one of the two nodes with smallest weight at the beginning of cycle $k+1$ and, as a consequence, it must be melded during this cycle, forming a new ancestor of s_*. Therefore, the result holds for $k+1$, what establishes the lemma. ■

As an immediate consequence of lemmas 1 and 2, we can state the following theorem.

Theorem 1 *The ParHuff algorithm runs in $O(H \log\log n)$ parallel time, with $O(nH)$ work.*

Proof: It follows immediately from lemmas 1 and 2. ■

3 The ES-ParHuff Algorithm

In this section we show how to modify the ParHuff algorithm to improve both its work and time complexity. For that, we need to parallelize a particular implementation of Huffman's algorithm proposed by Van Leeuwen [vL76]. We call the new algorithm *ES-ParHuff*.

ES-ParHuff improves the performance of ParHuff due two enhancements. First, it works with two lists. One for the internal nodes and other for the leaves. Second, it performs an exponential search [BY76] to decide which nodes must be melded in the current step. The algorithm runs in $O(H \log\log(n/H))$ parallel time, with $O(n)$ work. Before presenting ES-ParHuff, we briefly describe Van Leeuwen's implementation of Huffman's algorithm.

3.1 Van Leeuwen's Implementation

In this implementation, a stack and a queue are maintained. The stack for the leaves and the queue for the internal nodes. The nodes in both the stack and the queue are arranged in the non-decreasing order of their weights. Initially, the queue is empty and the stack contains all leaf nodes, with the one with smallest weight w_1 at its top. At each step, a new internal node y is formed. The two children of y are the ones with the smallest weights among four selected nodes, two from the current head of the queue and two from the current top of the stack. When ties occur, we choose leaf nodes. Each chosen node is removed and the corresponding top cursor updated for either the stack or the queue. After that, y is inserted in the queue. The process stops when just one node (the root of the tree) remains in the queue and the stack is empty.

3.2 Algorithm Description

Now, we describe the ES-ParHuff algorithm. A pseudo-code for the algorithm is presented in figure 4. The algorithm starts with a list S of leaves and a queue Q of internal nodes, both sorted by the non-decreasing order of their weights. At each cycle, the algorithm selects the two nodes with smallest weights among the leaves and the internal nodes. Next, it creates a parent t for them and uses the function **Select** to find the total number k of leaves with weights no greater than $w(t)$. The function **Select** is detailed in the next subsection. Next, the algorithm adds k to the current number of internal nodes in the queue. If the result is even, the leaves with weights no greater than $w(t)$ and all the current internal nodes are merged into the auxiliary list U. Otherwise, the weight $w(q_{length(Q)})$ of the last internal node of the queue is compared with $w(s_k)$. Then, all the nodes, but the one with the largest weight are merged into U. The algorithm assumes that the nodes involved in the merge operation are also removed from their respective lists. Finally, each pair of nodes u_{2i-1} and u_{2i} of the list U is melded to form a parent t_i that is inserted in Q. Again, a new cycle starts.

We observe that in the case of ties, the algorithm chooses a leaf. This choice implies that the resulting Huffman tree has also minimum height among all Huffman trees. This is very convenient since it would minimize the number of cycles of ES-ParHuff.

In the pseudo-code, we use S^j to denote the sublist of S starting with the first leaf and ending with j-th leaf. A similar notation is used for the queue Q of internal nodes. The function **Merge**$(S^j, Q^{j'})$ merges in parallel the sublist S^j of S with the sublist $Q^{j'}$ of Q.

3.3 Selecting the Leaves

The function **Select**$(S, w(t))$ finds the largest index k such that $w(s_k) \leq w(t)$. The search for s_k is performed in the following way. First, the algorithm uses one processor to perform an exponential search [BY76] in the set of indexes $2^0, 2^1, 2^2, \ldots, 2^{\lceil \log n \rceil}$ in order to find a value j such that $2^{j-1} < k \leq 2^j$. Next, $2^j - 2^{j-1} = 2^{j-1}$ processors are used to determine the value of k.

This search can be performed in $O(\log \log k)$ time, with $O(k)$ work as shows the next lemma.

ES-ParHuff Algorithm
$S \leftarrow$ sorted list of leaves ;$Q \leftarrow nil$
While $length(S) > 0$ or $length(Q) > 1$
Select the two nodes a and b with smallest weights in S or Q ;
Remove a and b from their corresponding lists ;
Create t_1 as a parent of a and b; $w(t_1) \leftarrow w(a) + w(b)$
$k \leftarrow$ **Select**$(S, w(t_1))$
If $k + length(Q)$ is even **then** $U \leftarrow$ **Merge**(S^k, Q);
Else
If $w(s_k) \leq w(q_{length(Q)})$ **then** $U \leftarrow$ **Merge**$(S^k, Q^{length(Q)-1})$;
Else $U \leftarrow$ **Merge**(S^{k-1}, Q);
End If
Insert t_1 in the queue Q;
For $i = 1$ to $length(U)/2$ **pardo**
Create t_i as a parent of u_{2i-1} and u_{2i}; $w(t_i) \leftarrow w(u_{2i-1}) + w(u_{2i})$
Insert t_i in the queue Q
End For
End While

Figure 4: A pseudo-code for ES-ParHuff algorithm

Lemma 3 *The function* ***Select****(S,w(t)) determines the largest value k such that $w(s_k) \leq w(t)$ in $O(\log\log k)$ parallel time, performing $O(k)$ work.*

Proof: The exponential search for j takes $O(\log j) = O(\log\log k)$ time. Next, the value of k is found in $O(1)$, with $2^{j-1} = O(k)$ processors. Hence, the function runs in $O(\log\log k)$ time, with $O(k)$ work. ■

3.4 Algorithm Analysis

In this section we analyze the ES-ParHuff algorithm. First, we determine both the time and work complexities of ES-ParHuff.

Theorem 2 *The ES-ParHuff algorithm runs in $O(H\log\log(n/H))$ time and its work is $O(n)$.*

Proof: First, we analyze the time complexity. Lemma 2 states that the number of cycles performed by the algorithm is H, where H is the height of the resulting Huffman tree.

At the i-th cycle, the exponential search spends $O(\log\log k(i))$, where $k(i)$ is the number of leaves removed from the list at the i-th cycle. Next, the algorithm performs some comparisons in $O(1)$ time. The merge operation can be performed in $O(\log\log I(i))$ parallel time, where $I(i)$ is the number of internal nodes formed at the i-th cycle. Finally, the nodes of U are melded in $O(1)$ time and the new internal nodes are inserted in Q also in $O(1)$ time. It follows that the algorithm spends

$$O(\sum_{i=1}^{H}(\log\log I(i) + \log\log k(i))) \tag{1}$$

time to construct the tree. Hence, an upper bound for the time spent by the algorithm can be obtained by maximizing (1) as a function of $I(i)$ and $k(i)$ for $i = 1, \ldots, H$ constrained to

$$\sum_{i=1}^{H} k(i) = n$$

and

$$\sum_{i=1}^{H} I(i) = n - 1.$$

By Jensen's inequality, the function in (1) is maximized when $k(i) = n/H$ and $I(i) = (n-1)/H$. Hence, the algorithm runs in $O(H \log\log(n/H))$ parallel time.

Now, we analyze the work. The exponential search performs $O(k(i))$ operations at the i-th cycle. Hence, the work due to the exponential search is also $O(n)$. On the other hand, the work for merging, melding and inserting operations is $O(I(i))$ at the i-th cycle. Hence, the work due these operations is $O(\sum_{i=1}^{H} I(i)) = O(n)$. Therefore, the total work performed by the algorithm is $O(n)$. ■

Since H is in the interval $[\lceil \log n \rceil, n-1]$, then in the worst case the algorithm is $O(n)$ and in the best case it is $O(\log n \log\log n)$. An extension for p processors is given in the following corollary.

Corollary 1 *Given an integer p, with $1 \leq p \leq n$, the ES-ParHuff algorithm runs in $O(n/p + H \log\log(n/H))$ parallel time using p processors.*

Proof: It follows immediately from theorem 2 and Brent's principle [Bre74]. ■

It is also possible to give a bound for the complexity time of ES-ParHuff as a function of the input weights as shows the next corollary.

Corollary 2 *The complexity time of ES-ParHuff is* $O(min(n, \log(1/p_1) \log\log n))$

Proof: In [Bur93], Buro proved for $w_1 \leq \ldots \leq w_n$ that

$$H \leq min\{\log_\phi(\phi + 1/(p_1\phi + p_2)), n-1\},$$

where ϕ is the golden ratio $(1+\sqrt{5})/2$ and $p_i = w_i / \sum_{j=1}^{n} w_j$. Since

$$\log_\phi(\phi + 1/(p_1\phi + p_2)) < \log_\phi(1/p_1),$$

it follows that the complexity time of ES-ParHuff is $O(min(n, \log(1/p_1) \log\log n))$. ■

4 Conclusions

In this paper we present ES-ParHuff, a work efficient PRAM CREW algorithm for constructing Huffman codes. For a sorted list of weights, this algorithm runs in $O(H \log\ \log(n/H))$ time with $O(n)$ work, where H is the length of the longest generated code. Since the best sequential implementation of Huffman's algorithm spends $O(n)$, then our parallel algorithm is work efficient. As far as we know, this is the first work efficient parallel algorithm for constructing Huffman codes. Furthermore, we prove that given $1 \leq p \leq n$, the ES-ParHuff algorithm runs in $O(n/p + H \log\log(n/H))$ parallel time using p processors.

We also prove that the complexity time of ES-ParHuff is $O(\min(n, \log(1/p_1) \log\log n))$, where $p_1 = w_1 / \sum_{i=1}^{n} w_i$. Hence, our algorithm overcomes Larmore and Przytcka's algorithm [LP95] when $\log(1/p_1)$ is $o(\sqrt{n} \log n / \log\log n)$ and Atallah's [AKL+89] algorithm when $\log(1/p_1)$ is $o(\log^2 n / \log\log n)$.

Finally, we observe that an important feature of ES-ParHuff is its simplicity. This could lead to very fast implementations that could be attractive for practical purposes.

References

[AKL+89] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S.-H. Teng. Constructing trees in parallel. In ACM-SIGACT; ACM-SIGARCH, editor, *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, page 421, Santa Fe, NM, June 1989. ACM Press.

[Bre74] Richrd P. Brent. The parallel evaluation of general arithmetic expessions. *Journal of the ACM*, 21(02):201–208, 1974.

[Bur93] Michael Buro. On the maximum length of Huffman codes. *Information Processing Letters*, 45(5):219–223, 2 April 1993.

[BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, August 1976.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[Col88] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 1(4):770–785, August 1988.

[Czu92] Artur Czumaj. Parallel algorithm for the matrix chain product problem. Research Report CS-RR-225, Department of Computer Science, University of Warwick, Coventry, UK, June 1992.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[Kru83] C. Kruskal. Searching, merging and sorting in parallel computations. *IEEE Trans. Comput.*, C-32(10):942–946, 1983.

[LP95] Lawrence L. Larmore and Teresa M. Przytycka. Constructing Huffman trees in parallel. *SIAM Journal on Computing*, 24(6):1163–1169, December 1995.

[MK95] Alistair Moffat and Jyrki Katajainen. In-place calculation of minimum-redundancy codes. In Selim G. Akl, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, 4th International Workshop*, volume 955 of *Lecture Notes in Computer Science*, pages 393–402, Kingston, Ontario, Canada, 16–18 August 1995. Springer.

[MM76] S. Michaelson and R. Milner, editors. *Third International Colloquium on Automata, Languages and Programming*, University of Edinburgh, 20–23 July 1976. Edinburgh University Press.

[MPL98] Ruy L. Milidiú, Artur A. Pessoa, and Eduardo S. Laber. A space-economical algorithm for minimum-redundancy coding. Technical Report 02, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brasil, January 1998.

[vL76] J. van Leeuwen. On the construction of Huffman trees. In Michaelson and Milner [MM76], pages 382–410.