algorithms

Article

Concurrent vs. Exclusive Reading in Parallel Decoding of LZ-Compressed Files [†]

Sergio De Agostino [1,*], Bruno Carpentieri [2] and Raffaele Pizzolante [2]

1 Computer Science Department, Sapienza University of Rome, Rome 00185, Italy
2 Dipartmento di Informatica, Università di Salerno, Fisciano (SA) 84084, Italy; bc@dia.unisa.it (B.C.); rpizzolante@unisa.it (R.P.)
* Correspondence: deagostino@di.uniroma1.it; Tel.: +39-06-4991-8355

† This paper is an extended version of our paper published in International Conference on Data Compression, Communication Processing and Security 2016.

Academic Editor: Spyros Kontogiannis
Received: 25 November 2016; Accepted: 23 January 2017; Published: 28 January 2017

Abstract: Broadcasting a message from one to many processors in a network corresponds to concurrent reading on a random access shared memory parallel machine. Computing the trees of a forest, the level of each node in its tree and the path between two nodes are problems that can easily be solved with concurrent reading in a time logarithmic in the maximum height of a tree. Solving such problems with exclusive reading requires a time logarithmic in the number of nodes, implying message passing between disjoint pairs of processors on a distributed system. Allowing concurrent reading in parallel algorithm design for distributed computing might be advantageous in practice if these problems are faced on shallow trees with some specific constraints. We show an application to LZC (Lempel-Ziv-Compress)-compressed file decoding, whose parallelization employs these computations on such trees for realistic data. On the other hand, zipped files do not have this advantage, since they are compressed by the Lempel–Ziv sliding window technique.

Keywords: LZ compression; decoding; pram; mapreduce

1. Introduction

Parallel random access machines (PRAMs) are out of fashion today, but an apology of this model can still be done from the point of view of parallel algorithm design. Indeed, the so-called PRAM model provides the most natural computational tool for a first approach to an algorithmic solution of a problem with parallel computing, and secondly, a distributed memory implementation can be derived from it on a network. The computing techniques involved in the design of parallel and distributed algorithms strictly relate to the computational model on which the parallel or distributed system is based. The efficiency of a technique designed for a specific model can consistently deteriorate when applied to a different system. This is particularly evident when a technique designed for a shared memory parallel random access machine is implemented on a distributed system. Indeed, when the system is scaled up, the communication cost is a bottleneck to linear speed-up. So, we need to limit the interprocessor communication, either involving more local computation or bounding the number of global computation steps in order to obtain a practical algorithm. Local computation might cause a lack of robustness when scalability properties are required. On the other hand, scalability and robustness are generally guaranteed if bounding the number of global computation steps is possible for a specific problem.

In this paper, we face the issue of concurrent versus exclusive reading in the design of a parallel algorithm for message passing-based distributed computing with an application to Lempel–Ziv data compression [1–3]. Broadcasting a message from one to many processors in a network corresponds to concurrent reading on a random access shared memory parallel machine, while exclusive reading implies message passing between disjoint pairs of processors on a distributed system. Basic subroutines for parallel algorithm design are computing the trees of a forest, the level of each node in its tree, and the path between two nodes. These problems can easily be solved with concurrent reading in a time logarithmic in the maximum height of a tree by means of the pointer jumping technique. The well-known Euler tour technique developed in [4] avoids concurrent reading by a linearization of the forest, reducing these problems to list ranking and parallel prefix. Solving such problems with exclusive reading requires a time logarithmic in the number of nodes. Besides avoiding concurrent reading, the Euler tour technique makes the parallel computation of several tree functions possible, as preorder, post-order, and computing the number of descendants of each node in logarithmic time with a linear number of processors. Without such linearization, these functions cannot be computed with such parallel complexity, even if concurrent reading is allowed. On the other hand, allowing concurrent reading to compute the trees of a forest, the level of each node in its tree, and the path between two nodes might be advantageous in parallel algorithm design for distributed computing in practice if such problems are faced on shallow trees with some specific constraints. Indeed, the concurrent reading is caused by the simultaneous access of two or more children to the information stored in a parent node, which increases at each step with the descendants. Therefore, the slow-down of such concurrency would not be so relevant if limited to a constant number of nodes. We evidentiate the properties a tree must have to obtain such limitation, and observe how—in the case of shallow trees—applying the Euler tour technique is not only unnecessary, but even disadvantageous for the running time.

We show an application to LZC compressed file decoding [5,6], whose parallelization employs these computations on such trees for realistic data. The pair compressor/decompressor presents an asymmetry with respect to global parallel computation, since the encoder is not parallelizable [7,8], while the decoder has a very efficient parallelization [8–10]. On realistic data, the number of iterations of the decoding algorithm is much less than ten units when expressed on the parallel random access shared memory machine [10], and about ten units when expressed in the MapReduce parallel programming paradigm [11], as we will show in this paper. This makes it attractive in those cases (which are the most common in practice) where compression is performed very rarely, while the frequent reading of raw data needs fast decompression. On the other hand, zipped files do not have such advantage, since they are compressed by the Lempel–Ziv sliding window technique [2,12,13]. In such cases, both coding and decoding are parallelizable in a symmetric way, requiring at least logarithmic time in practice [8,14].

In Section 2, we describe the computing techniques involved with the design of a parallel algorithm and discuss the concurrent versus exclusive reading issue. Section 3 shows the application to parallel decompression. The MapReduce implementation is described in Section 4. Conclusions and future work are given in Section 5.

2. Concurrent versus Exclusive Reading

If we use a parent array as data structure to represent a forest, computing the trees, the level of each node in its tree, and the path between two nodes are problems which can easily be solved on a CREW (concurrent read, exclusive write) PRAM by running in parallel the pointer jumping operation $parent[i] = parent[parent[i]]$. The procedure takes a number of processors linear in the number of nodes and a time logarithmic in the maximum height of a tree. Generally speaking, implementing such a procedure causes a slow-down logarithmic in the number of nodes if we solve the reading conflicts by standard broadcasting techniques. Therefore, the total running time increases to square-logarithmic in the worst case, since the maximum height of a tree can reach the order of magnitude of the number of nodes. To keep the time logarithmic, we need to assume some constraints on the structure of each tree of the forest. Indeed, there must be a constant $c > 0$ such that for each tree of the forest, the number of nodes with more than one child distant more than c from the root is less than c, and none of the nodes more distant than c from the root has more than c children. It is easy to see that this is a necessary and sufficient condition to solve the reading conflicts by standard broadcasting with no slow-down on the total running time in terms of asymptotic behavior of the parallel complexity. If such a constant does not exist, the well-known Euler tour technique developed in [4] avoids the concurrent reading of the children.

The Euler tour technique reduces the problems to list ranking on the EREW (exclusive read, exclusive write) PRAM with a number of processors and a time, respectively, linear and logarithmic both in the number of nodes. Let F be a forest of rooted trees with every node having a parent pointer along with a doubly-linked list of children. To begin, we replace every node v of F by an array $V[1 \cdots d(v) + 1]$ of copies of v, where $d(v)$ is the number of children of v. Let $w_1 \cdots w_{d(v)}$ be the children of v and $W_1[1 \cdots d(w_1) + 1], \cdots W_{d(v)}[1 \cdots d(w_{d(v)} + 1)]$ the corresponding arrays. We link $W_i[1]$ to $V[i]$ and $V[i + 1]$ to $W_i[d(w_i) + 1]$, and obtain for each tree T of F a linked list starting at $R[1]$ and ending at $R[d(r) + 1]$, where r is the root of T and R the corresponding array. Then, it is clear that we find the trees in the forest by applying list ranking. Moreover, for each list associated with a tree, we assign to the first entry in every array a weight +1, and to all the others -1. Then, the level of each node in its tree is obtained with prefix computation by computing the partial sums of these values in the list. Partial sums corresponding to positions of components of the array associated with v in the list are equal to the level of v in its tree. Since we assumed to have children doubly linked with their parent, the realization of the list takes constant time with a linear number of processors or logarithmic time with O($n/\log n$) processors. List ranking and prefix computation require logarithmic time with O($n/\log n$) processors as well. If children are not doubly linked to their parent, adding the links from a parent to its children generally takes logarithmic time with a linear number of processors employing parallel sorting procedures. However, there are cases where the bound to the number of children is known and constant.

To conclude this section, we wish to point out that even if there is a constant bounding the nodes with more children and the children of a node too distant from the root, the Euler tour technique might still be advantageous when the tree is not shallow enough. However, this is not the case for the application to LZC-compressed file decoding that we present in this paper.

3. Decoding LZC-Compressed Files in Parallel

The most popular text compressors are based on Lempel–Ziv string complexity [1]. Zip compressors apply the Lempel–Ziv sliding window method [2,12,13], while other applications use the LZW (Lempel, Ziv, and Welch) compressor [3,5], also called LZC [6] when implemented as the command line "compress" does on Unix and Linux platforms. LZC compression is less effective but faster than the zipping applications. Zip compressors are based on a string factorization process, where each factor is independent from the others since it extends by one character the longest match with a substring to its left in the input string. On the other hand, the LZC compressor is based on a factorization, where each factor is the extension by one character of the longest match with one of the previous factors. This computational difference implies that while the zipping applications have theoretical parallelizations, LZC compression is hard to parallelize. On the other hand, parallel decompression is possible for both approaches.

Zipping and unzipping files in parallel requires a logarithmic number of global computation steps, even if concurrent reading is allowed. On the other hand, the LZC encoder/decoder pair presents an asymmetry, since the decoder has a very efficient parallelization requiring much less than ten iterations on the CREW PRAM. As mentioned above, this makes LZC more attractive than Zip in those cases (which are the most common in practice) where compression is performed very rarely while the frequent reading of raw data needs fast decompression. Finally, we show that this advantage is lost if the LZC decoder is implemented on the EREW PRAM.

3.1. LZC Compression

Lempel–Ziv compression is a dictionary-based technique. Indeed, the factors of the string are substituted by *pointers* to copies, stored in a *dictionary*, which are called *targets*. In practical implementations, the dictionary has a fixed size $d + \alpha$, where α is the cardinality of the alphabet. With LZC, $d + \alpha = 2^{16}$, and the dictionary is initially equal to the alphabet. The LZC factorization of a string S fills up the dictionary by factorizing a prefix P of S. The factorization $P = f_1 f_2 \cdots f_i \cdots f_d$ is such that f_i is either the longest match with the concatenation of a previous factor f_j and the next character or the current alphabet character if there is no match (that is, every factor concatenated with the next character is a dictionary element). The filled up dictionary is "frozen", and the factorization continues in a "static" way for a while; that is, for $d < i \le t$, the factor f_i is either the longest match with the concatenation of a previous factor f_j, with $j \le d$, and the next character or the current alphabet character if there is no match. The value t is determined by monitoring the compression ratio. When this ratio deteriorates, the dictionary is reset to be equal to the alphabet. LZC compression is the one employed by the Unix and Linux "compress" applications, and similar applications have been realized with Stuffit on Windows and Dos platforms.

3.2. The CREW PRAM Algorithm

A special mark occurs in the sequence of pointers of the LZC encoding when the dictionary is cleared out, so that the decoder does not have to monitor the compression ratio. The positions of the special mark are detected by parallel prefix. Each subsequence $q_1 \cdots q_m$ of pointers between two consecutive marks can be decoded in parallel. The parallel algorithm is based on the fact that the target of the pointer q_i in the subsequence is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, and employs concurrent reading [9].

Let $Q_1 \cdots Q_N$ be the LZC encoding of a string S, drawn over an alphabet A of cardinality α, with Q_h sequence of pointers between two consecutive reset operations, for $1 \le h \le N$. Each Q_h can be decoded independently. Let $q_1 ... q_m$ be the sequence of pointers Q_h encoding the substring S' of S. The decoding of Q_h requires on a CREW PRAM $O((log(L))$ time with $O(|S'|)$ processors, where L is the maximum length of a pointer target. The dictionary size is a theoretical upper bound to L that is tight for unary strings. The algorithm computes an *$m \times d$ matrix of prefix pointers M*, initially null with $d = 2^{16}$, as it follows:

```
input: sequence of pointers $q_1 ... q_m$;
output: matrix $M$ of prefix pointers;
1. $k := 1$;
2. **in parallel for** $1 \le i \le m$ **do**
3.   $M[i,j] := q_i$;
4.   $last[i] := 1$;
5.   $value[i] := M[i,j] - d$;
6.   **while** $value[i] > 0$ **do**
7.     **in parallel for** $1 \le j \le k$ **do**
8.       **if** $j \le last[value[i]]$ **then**
9.         $M[last[i] + j.i] = M[j, value[i]$;
10.      $last[i] := last[i] + last[value[i]]$;
11.      $value[i] := value[value[i]]$;
13.      $k := 2k$;
```

At each step, $last[i]$ is the last nonnull component on the i^{th} column considered (line 10 after the initialization at line 4). The nonnull components of the $value(i)^{th}$ column are copied on the i^{th} column in the positions after $last[i]$ (lines 7–9). Note that $value(i)$ is strictly less than i. Then, $value(i)$ is updated by setting $value(i) := value(value(i))$ (line 11). The loop stops on column i when $value(i)$ is less or equal to zero. This procedure takes $O(|S'|)$ processors and $O(log(L))$ time on a CREW PRAM, since the number of nonnull components on a column doubles at each step. The target of the pointer q_i is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, since the dictionary initially contains the alphabet characters. At the end of the procedure, $M[last[i], i]$ is the pointer representing the first character of the target of q_i and $last[i]$ is the target length. Then, we conclude that $M[last[M[j,i] - d + 1], M[j,i] - d + 1]$, for $1 \leq j \leq last[i] - 1$, is the pointer representing the $(last[i] - j + 1)^{th}$ character of the target of q_i. That is, we have to look at the pointer values written on column i and consider the last nonnull components of the columns in the positions given by such values decreased by $\alpha - 1$. Such components must be concatenated according to the bottom-up order of the respective values on column i. By mapping each component into the correspondent alphabet character, we obtain the suffix following the first character of the target of q_i, and the pointers are therefore decoded (see [9] for further clarifications).

The sequence of pointers can be seen as a parent pointer representation of a forest, where each pointer q with a value greater than α represents a link from a node associated with q to its parent node associated with the pointer in position $q - \alpha$. The making of the matrix computes on each column the path from the node associated with the corresponding pointer to the root of its tree. Such root is always associated to an alphabet character. The maximum height of a tree is the maximum length of a factor. The theoretical upper bound to the factor length is the dictionary size, which is tight in the unary string case. However, on realistic data we can assume that the maximum factor length L is such that $10 < L < 20$. The motivation for this assumption is that in practice, the maximum length of a factor is much smaller than the dictionary size. For example, when compressing english text with sixteen-bit pointers, the average match length will only be about five units (for empirical results, see [13]). It follows that the number of iterations (global computation steps) is much less than ten if the PRAM CREW algorithm were executed in practice. In some exceptional cases, the maximum factor length will reach one hundred units; that is, the number of iterations will be equal to seven units. Moreover, the number of children for each node is theoretically bounded by the alphabet cardinality, but in practice, such bound works only for the root. After a few levels, it is realistic to assume that nodes have only one child. So, for realistic data there is a small constant $c > 0$ such that for each tree of the forest, the number of nodes with more than one child distant more than c from the root is less than c, and none of the nodes distant more than c from the root has more than c children. Therefore, concurrent reading is resolved by standard broadcasting techniques with no relevant slow-down.

3.3. The EREW PRAM Algorithm

The subsequence $q_1 \cdots q_m$ of pointers between two consecutive reset operations that we decoded on the CREW PRAM in the previous subsection is decoded on the EREW PRAM in two phases. In the first phase, since the pointers do not contain the information on the length of their targets, these lengths have to be computed. The target of the pointer q_i in the subsequence is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, where α is the alphabet cardinality. Then, in parallel for each i, link pointer q_i to the pointer in position $q_i - \alpha$ if $q_i > \alpha$. Again, we obtain a forest where each tree is rooted in a pointer representing an alphabet character, and the length l_i of the target of a pointer q_i is equal to the level of the pointer in the tree plus 1. It is known from [1] that the largest number of distinct factors whose concatenation forms a given string of length ℓ is $O(\ell / \log \ell)$. Since a factor of the LZW factorization of a string appears a number of times which is at most equal to the alphabet cardinality, it follows that m is $O(\ell / \log \ell)$ if ℓ is the length of the substring encoded by the subsequence $q_1 \cdots q_m$. Then, building such a forest takes $O(log(|S'|))$ time with $O(|S'|)$ processors on a shared memory parallel machine without writing and reading conflicts. With the same parallel complexity, we can compute the trees of such forest and the level of each node in its own tree by means of the Euler tour technique. Therefore, we can compute the lengths $l_1, ..., l_m$ of the targets. If $s_1, ..., s_m$ are the partial sums, the target of q_i is the substring over the positions $s_{i-1}+1\cdots s_i$ of the output string. For each q_i which does not correspond to an alphabet character, define $first(i) = s_{q_i-\alpha-1}+1$ and $last(i) = s_{q_i-\alpha}+1$. Since the target of the pointer q_i is the concatenation of the target of the pointer in position $q_i-\alpha$ with the first character of the target of the pointer in position $q_i-\alpha+1$, link the positions $s_{i-1}+1\cdots s_i$ to the positions $s_{first(i)}\cdots s_{last(i)}$, respectively. As in the sliding dictionary case, if the target of q_i is an alphabet character, the corresponding position in the output string is the root of a tree in a forest, and all the nodes in a tree correspond to positions of the decoded string where the character is the root. Since the number of children for each node is at most α, in $O(log(|S'|))$ time with $O(|S'|)$ processors, we can store the forest in a doubly linked structure and decode by means of the Euler tour technique on the EREW PRAM [14]. Differently from the CREW PRAM algorithm, the running time is obviously $O(log(|S'|))$, regardless of the factor maximum length.

4. The MapReduce Implementation of the CREW Algorithm

We show how to implement the CREW PRAM algorithm of the previous section in MapReduce, and discuss the complexity issues. First, we present the MapReduce model of computation.

4.1. The MapReduce Model of Computation

The MapReduce model allows global computation on a distributed system in its theoretical formulation. Therefore, bounding the number of computational steps is a requirement for the design of a practical algorithm.

The MapReduce programming paradigm is a sequence $P = \mu_1\rho_1\cdots\mu_R\rho_R$, where μ_i is a mapper and ρ_i is a reducer for $1 \le i \le R$. First, we describe this paradigm and then discuss how to implement it on a distributed system. Since the input/output phases are inherent to any parallel algorithm and have standard solutions, the sequence P does not include the I/O phases, and the input to μ_1 is a multiset U_0 where each element is a $(key, value)$ pair. The input to each mapper μ_i is a multiset U_{i-1} output by the reducer ρ_{i-1}, for $1 < i \le R$. Mapper μ_i is run on each pair (k, v) in U_{i-1}, mapping (k, v) to a set of new $(key, value)$ pairs. The input to reducer ρ_i is U'_i, the union of the sets output by μ_i. For each key k, ρ_i reduces the subset of pairs of U'_i with the key component equal to k to a new set of pairs with key component still equal to k. U_i is the union of these new sets.

In a distributed system implementation, a key is associated with a processor. All the pairs with a given key are processed by the same node, but more keys can be associated to it in order to lower the scale of the system involved. Mappers are in charge of the data distribution, since they can generate new key values. On the other hand, reducers just process the data stored in the distributed memory, since they output for a set of pairs with a given key another set of pairs with the same given key.

The following complexity requirements are stated as necessary for a practical interest in [11]:

- R is polylogarithmic in the input size n;
- the number of processors (or nodes in the Web) involved is $O(n^{1-\epsilon})$ with $0 < \epsilon < 1$;
- the amount of memory for each node is $O(n^{1-\epsilon})$;
- mappers and reducers take polynomial time in n.

In [11], it is also shown that a $t(n)$ time CREW PRAM algorithm using subquadratic work space and a subquadratic number of processors can be implemented by MapReduce with a simulation satisfying the above requirements if $t(n)$ is polylogarithmic. Indeed, the parameter R of the simulation is $O(t(n))$, while the subquadratic work space is partitioned among a sublinear number of processors taking polynomial computational time.

Such requirements are necessary but not sufficient to guarantee a speed-up of the computation. Obviously, the total running time of mappers and reducers cannot be higher than the sequential one, and this is trivially implicit in what is stated in [11]. The non-trivial bottleneck is the communication cost of the computational phase. This needs to be checked experimentally, since R can be polylogarithmic in the input size. Generally speaking, a MapReduce implementation has a practical interest if R is about ten units or less. If this is obtained from the simulation of a CREW PRAM algorithm, it might be preferable to the simulation of an EREW PRAM algorithm with a higher number of iterations.

4.2. Decoding LZC-Compressed Files in MapReduce

The MapReduce implementation of the decoder decompressing the sequence of pointers $q_1 ... q_m$ of the previous section is $P = \mu_0\rho_0\mu_1\rho_1 \cdots \mu_{2R}\rho_{2R}\mu_{2R+1}\rho_{2R+1}$, with $R = \lceil logL \rceil$. The number of iterations is $2R + 2$ since, generally speaking, the simulation of a CREW PRAM algorithmic step is realized by two mappers and two reducers, where the reducers compute the memory requests and the corresponding information that must be provided to the processors while the mappers route the memory requests and the information to the reducers responsible for the particular processor [11]. In this particular case, the keys correspond to the matrix entries, and the reducers compute the memory requests by looking at the values associated with the keys corresponding to the matrix entries, storing the last non-null components on the columns. As far as the other reducers are concerned, the information that must be provided to the processors is already computed, since the procedure just consists of copying values from columns to columns. Therefore, such reducers just identify the processors (or the keys) for the mappers routing the information.

The input to μ_0 is a multiset U_{-1} of cardinality m, where each element is a $(key, value)$ pair with $key = (1, i)$ and $value = q_i$ for $1 \leq i \leq m$. The output of μ_0 is $U'_0 = U_{-1} \cup O'_0$, where each element in O'_0 is a $(key, value)$ pair with $key = (1, i)$ and $value = i'$ such that $i < i' \leq m$ and $q_{i'} - \alpha = i$. Then, reducer ρ_0 outputs the set $U_0 = U_{-1} \cup O_0$, where O_0 is obtained from O'_0 by reducing each element $((1, i), i')$ to the element $((1, i), (2, i'))$. In other words, μ_0 (as every other mapper of the sequence with an even index) routes a memory request for every processor. Since μ_0 is the first mapper, it also does the job of computing the memory request (that is, subtracting the alphabet cardinality to the pointer value). Afterwords, this job is done by each reducer with an odd index for the next mapper. Reducer ρ_0 (as every other reducer with an even index) computes the keys that the next mapper will use to route the information.

The keys computed by ρ_0 are used by mapper μ_1. The output of μ_1 is $U'_1 = U_{-1} \cup O'_1$, where each element in O'_1 is a $(key, value)$ pair with $key = (2, i')$ and $value = q > 0$ such that $((1, i), (2, i')) \in O_0$ and $q_i = q$. Then, reducer ρ_1 outputs the set $U_1 = U'_1 \cup O_1$, where O_1 is the set of elements with key $(2, i)$ and value $q - \alpha$ such that $((2, i), q) \in U'_1$ and $q - \alpha > 0$. So, reducer ρ_1 does the job that μ_0 did by itself. Therefore, mapper μ_2 operates in a slightly different way from μ_0 as every other mapper with an even index.

Mapper μ_2 outputs $U'_2 = U'_1 \cup O'_2$, where each element in O'_2 is a $(key, value)$ pair with $key = (1, i)$ or $key = (2, i)$ and $value = i'$ such that $i < i' \leq m$ and $q_{i'} - \alpha = i$, that is, $((2, i'), i) \in O_1$. Then, reducer ρ_2 outputs the set $U_2 = U'_1 \cup O_2$, where O_2 is obtained from O'_2 by reducing each element $((1, i), i')$ or $((2, i), i')$ in O'_2 to the element $((1, i), (3, i'))$ or $((2, i), (4, i'))$. To complete the first two CREW PRAM algorithmic steps, we describe mapper μ_3 and reducer ρ_3.

Mapper μ_3 outputs $U'_3 = U'_1 \cup O'_3$. Each element in O'_3 is a $(key, value)$ pair with $key = (3, i')$ or $key = (4, i')$ and $value = q > 0$ such that $((1, i), (3, i')) \in O_2$ or $((2, i), (4, i')) \in O_2$ and $q_i = q$. Then, reducer ρ_3 outputs the set $U_3 = U'_3 \cup O_3$, where O_3 is the set of elements with key $(4, i)$ and value $q - \alpha$ such that $((4, i), q) \in U'_3$ and $q - \alpha > 0$, similarly to ρ_1. Now, we can provide the MapReduce implementation of the generic step.

At the k-th step, for $4 \leq k \leq 2R - 1$, if k is even mapper μ_k outputs $U'_k = U'_{k-1} \cup O'_k$. Each element in O'_k is a $(key, value)$ pair with key (χ, i), for $1 \leq \chi \leq t$ and $t \leq 2^{k/2-1}$, and value i' such that $i < i' \leq m$ and $q_{i'} - \alpha = i$; that is, $((2^{k/2-1}, i'), i) \in O_{k-1}$. Then, reducer ρ_k outputs the set $U_k = U'_{k-1} \cup O_k$, where O_k is obtained from O'_k by reducing each element $((\chi, i), i') \in O'_2$ to the element $((\chi, i), (2^{k/2-1} + \chi, i'))$.

If k is odd, mapper μ_k outputs $U'_k = U'_{k-2} \cup O'_k$. Each element in O'_k is a $(key, value)$ pair with $key = (j, i')$ and $value = q > 0$ such that $(j - \chi, i), (j, i')) \in O_{k-1}$ and $q_i = q$ for some χ with $1 \leq \chi \leq t$

and $t \leq 2^{k/2-1}$. Then, if $k < 2R - 1$, reducer ρ_k outputs the set $U_k = U'_k \cup O_k$, where O_k is the set of elements with key $(2^{\lceil k/2 \rceil}, i)$ and value $q - \alpha$ such that $((2^{\lceil k/2 \rceil}, i), q) \in U'_k$ and $q - \alpha > 0$.

Reducer ρ_{2R-1} outputs the set $U_{2R-1} = U'_{2R-1} \cup O_{2R-1}$, where each $((j,i), q) \in U'_{2R-1}$ is reduced to $((j,i), q - \alpha + 1) \in O_{2R-1}$. Mapper μ_{2R} outputs $U'_{2R} = U'_{2R-1} \cup O'_{2R}$. Each element in O'_{2R} is either equal to $((1,i), q)$, where $((\ell_i, i), q) \in U'_{2R-1}$ and ℓ_i is the length of the factor encoded by q_i or to a $(key, value)$ pair with key equal to $(1, i)$ and value (j', i') such that $((j', i'), i) \in O_{2R-1}$. Then, reducer ρ_{2R} outputs U_{2R}, where each element $((1,i), (q, (j', i'))) \in U_{2R}$ is obtained from the two elements $((1,i), q)$ and $((1,i), (j', i'))$ in U'_{2R}.

Finally, μ_{2R+1} outputs U'_{2R+1} by mapping the element $((1,i), (q, (j', i')))$ to the element $((1,i'), (j', a)) \in U'_{2R+1}$, where a is the alphabet character target of q. Then, reducer ρ_{2R+1} outputs U_{2R+1} by reducing the set of elements $\{((1,i'), (a, j')) \in U'_{2R+1} : 1 \leq j' \leq \ell_{i'}\}$ to the element $((1,i'), f_{i'})$, where $f_{i'}$ is the target of $q_{i'}$ obtained by concatenating each alphabet character a for $\ell_{i'} \geq j' \geq 1$.

4.3. Complexity Issues

If $Q^1 \cdots Q^N$ is the encoding of the input string S, with $Q^h = q_1^h ... q_{m^h}^h$ sequence of pointers between two consecutive reset operations for $1 \leq h \leq N$, the MapReduce decoding sequence $P = \mu_0 \rho_0 \mu_1 \rho_1 \cdots \mu_{2R} \rho_{2R} \mu_{2R+1} \rho_{2R+1}$ of Q^h is such that $2R + 2$ is about ten units in practice. It is easy to extend the MapReduce sequence P to a MapReduce sequence $\Pi = M_0 P_0 M_1 P_1 \cdots M_{2R} P_{2R} M_{2R+1} P_{2R+1}$, where the input to M_0 is the multiset $\cup_{h=1}^{N} V^h$ and each element in V^h is a $(key, value)$ pair with $key = (1, i)$ and $value = q_i^h$, for $1 \leq i \leq m^h$ and $1 \leq h \leq N$. For $1 \leq h \leq N$, the mappers and reducers of the sequence Π operate as described in the previous subsection on the pointers corresponding to h. Let $H = \max\{m^h : 1 \leq h \leq N\}$. The sub-linearity requirements stated in [11] are satisfied if we implement Π on a cluster of H processors, since N and m^h, for $1 \leq h \leq N$, are generally sub-linear in practice. Moreover, the time of a MapReduce operation multiplied by the number of processors is $O(T)$, with T sequential time, since $2R + 2$ is about ten units (optimality requirement). This makes this MapReduce implementation of practical interest, since it has a small number of iterations. Finally, concurrent reading is resolved by standard broadcasting techniques with no relevant slow-down, since the MapReduce sequence implements the CREW PRAM algorithm of the previous section.

5. Conclusions

In this paper, we discussed when the design of a concurrent read, exclusive write parallel algorithm might have more practical value than the corresponding exclusive read version. We showed a practical example with a MapReduce implementation for decoding LZC compressed files. As future work, we would like to implement the MapReduce implementation of the LZC decoder on today's large scale clusters.

Author Contributions: Bruno Carpentieri and Raffaele Pizzolante conceived how to implement in practice the parallel decoding algorithm designed by Sergio De Agostino; Sergio De Agostino wrote the MapReduce implementation and the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lempel, A.; Ziv, J. On the Complexity of Finite Sequences. *IEEE Trans. Inf. Theory* **1976**, *22*, 75–81.
2. Lempel, A.; Ziv, J. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–443.
3. Ziv, J.; Lempel, A. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536.
4. Tarjan, R.E.; Vishkin, U. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. Comput.* **1985**, *14*, 862–874.
5. Welch, T.A. A Technique for High-Performance Data Compression. *IEEE Comput.* **1984**, *17*, 8–19 .
6. Bell, T.C.; Cleary, J.G.; Witten, I.H. *Text Compression*; Prentice Hall: Upper Saddle River, NJ, USA, 1990.
7. De Agostino, S. P-complete Problems in Data Compression. *Theor. Comput. Sci.* **1994**, *127*, 181–186.

8. De Agostino, S. Lempel-Ziv Data Compression on Parallel and Distributed Systems. *Algorithms* **2011**, *4*, 183–199.
9. De Agostino, S. A Parallel Decoding Algorithm for LZ2 Data Compression. *Parallel Comput.* **1995**, *21*, 1957–1961.
10. De Agostino, S. The Uncompress Application on Distributed Communications Systems. In Proceedings of the ICNS2015: The Eleventh International Conference on Networking and Services, Rome, Italy, 24–29 May 2015; pp. 55–60.
11. Karloff, H.J.; Suri, S.; Vassilvitskii, S. A Model of Computation for MapReduce. In Proceedings of the SIAM-ACM Symposium on Discrete Algorithms, Austin, TX, USA, 17–19 January 2010; pp. 938–948.
12. Storer, J.A.; Szymansky, T.G. Data Compression via Textual Substitution. *J. ACM* **1982**, *24*, 928–951.
13. Storer, J.A. *Data Compression: Methods and Theory*; Computer Science Press: Rockville, MD, USA, 1988.
14. De Agostino, S. Almost Work-Optimal PRAM EREW Decoders of LZ-Compressed Text. *Parallel Process. Lett.* **2004**, *14*, 351–359.