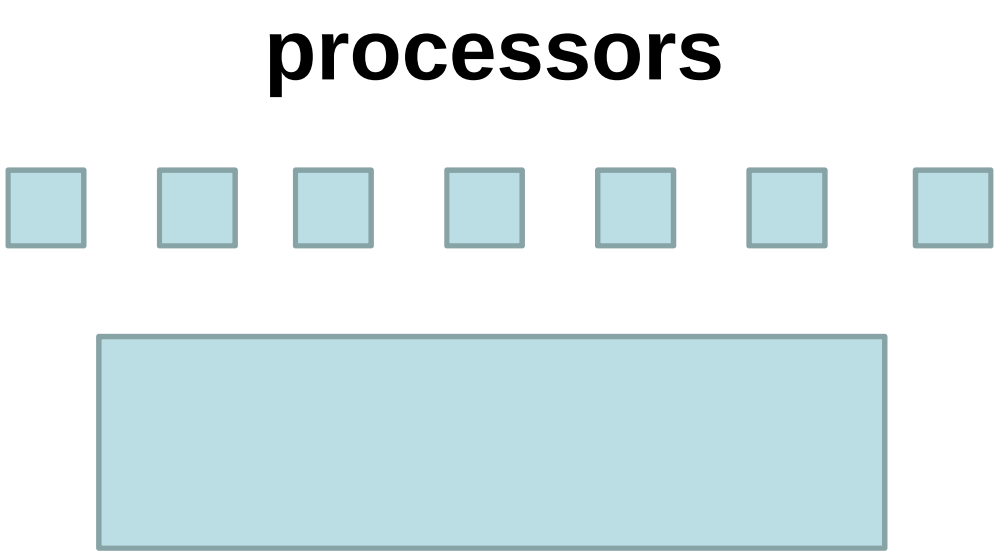
Computing Techniques for Parallel and Distributed Systems with an Application to Data Compression

Sergio De Agostino
Sapienza University di Rome

Parallel Systems

A parallel random access machine (PRAM) is a computational model with many processors and a shared memory.



EREW and CREW PRAMs

An EREW (exclusive read, exclusive write) PRAM is a machine where no memory access conflict is allowed.

When a PRAM algorithm is designed, it might come “natural” to allow cuncurrent reading (CREW) which can be managed by specific techniques.

Distributed Systems

Exclusive read on the PRAM implies message passing between disjoint pairs of processors on a distributed system.

Brodcasting a message from one to many processors corresponds to a cuncurrent read on the PRAM.

The possible slowdown, moving from a shared memory to a distributed one, depends on the network topology.

Outline Part 1

Parallel Computing Techniques:

- prefix computation
- list ranking
- pointer jumping
- Euler tour

Applications to file zipping and unzipping.

Distributed implementations.

Outline Part 2

LZW compression.

Parallel complexity issues.

Distributed Implementations.

Parallel decompression.

LZW decompression and MapReduce.

Conclusion.

Questions.

Prefix Computation

Let S be an ordered set $\{s_1, \ldots, s_n\}$ with an *associative* operation $\bullet$.

A *prefix computation* calculates:

s_1

$s_1 \bullet s_2$

$s_1 \bullet s_2 \bullet s_3$

……

$s_1 \bullet s_2 \bullet \ldots \bullet s_{n-1} \bullet s_n$

Parallel Prefix

Store S into an array S[] (s_i = S[i])

Parallel algorithm:

for j := 0 **to** lg(n) - 1 **do**
    for i := 2^j +1 **to** n **parallel-do**
        S[i] := S[i-2^j] ● S[i]

The algorithm performs the prefix computation in logarithmic time with a linear number of processors on an EREW PRAM.

Prefix Sum

S= {7, 6, 1, 8, 5, 3, 1, 9} with ● = + .

7	6	1	8	5	3	1	9
7	13	7	9	13	8	4	10
7	13	14	22	20	17	17	18
7	13	14	22	27	30	31	40

Prefix sums are computed.

Prefix Minimum

$x \bullet y = \min\{x, y\}$ is such that $(x \bullet y) \bullet z = x \bullet (y \bullet z)$.

- $\bullet$ is associative.

The prefix minimum on the ordered set

S= {7, 6, 1, 8, 5, 3, 1, 9} is:

{7, 6, 1, 1, 1, 1, 1, 1}

It can be computed with the same algorithm used for the prefix sums.

List Ranking

Linear list: b → a → b → c → e

List ranking: 1 2 3 4 5

List ranking can be reduced to prefix sum:

- associate a 1 to each element of the list;

- compute *ranks* with prefix sums.

List ranking is linear *pointer jumping*

Pointer Jumping

Find the roots in a forest of rooted trees.

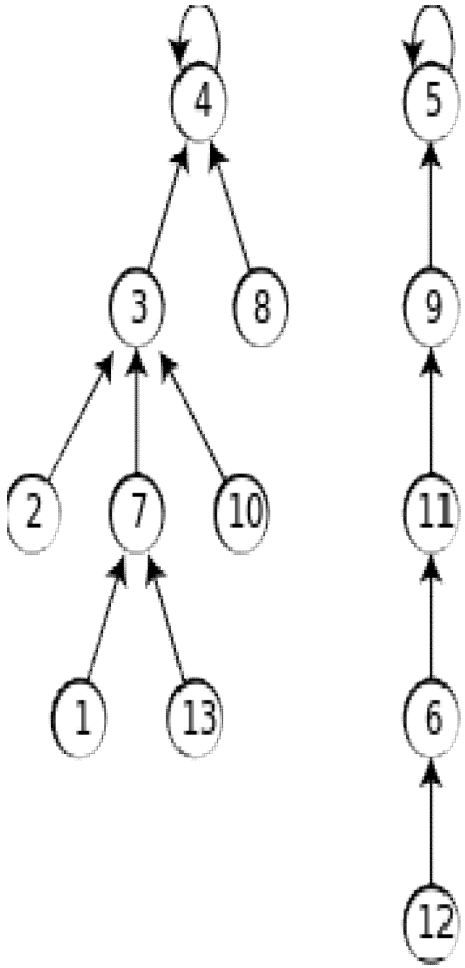
Use a *parent array* as data structure.

Jumping procedure with cuncurrent read:

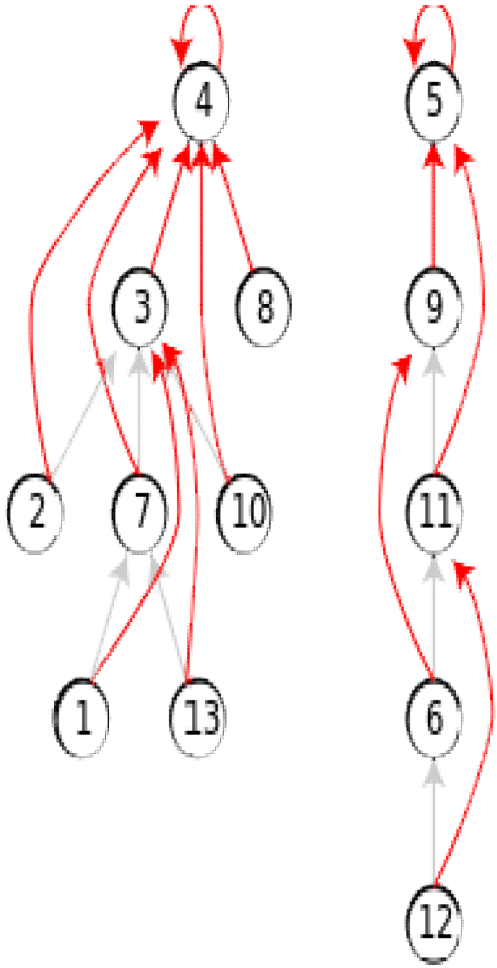
for i := 1 **to** n **do in parallel**
while parent[parent[i]] > 0
parent[i] ← parent[parent[i]]

Each node knows its root in *O(log n)* time with *O(n)* processors (*n* nodes number).

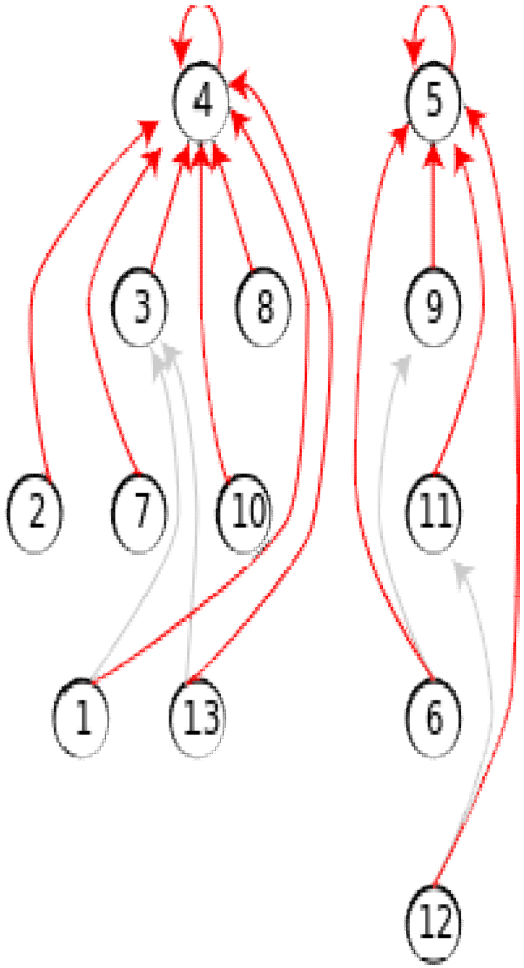
Example



Initial Forrest



First Iteration



Second Iteration

Computing the Paths

O(nh) processors compute each path from a node to its root in *O(log h)* time with *h* maximum height of a tree.

Build an *n* x *h* matrix by substituting:

parent[i] ← parent[parent[i]]

with a copy and append operation from the column indexed by the last element if it is greater than zero.

Example

column:	1	2	3	4	5	6	7	8	9
step 0:	0	1	0	2	3	5	6	5	3
step 1:	0	1	0	2	3	5	6	5	3
		0		1	0	3	5	3	0
step 2:	0	1	0	2	3	5	6	5	3
		0		1	0	3	5	3	0
				0		0	3	0	
							0		

Euler Tour Technique

Reduce pointer jumping to list ranking.

F forest of doubly linked rooted trees.

v node in the forest.

d(v) number of children of *v*.

Replace each node *v* by V[1…d(v)+1].

Each element of the array V is a copy of *v*.

Making a Linear Forest

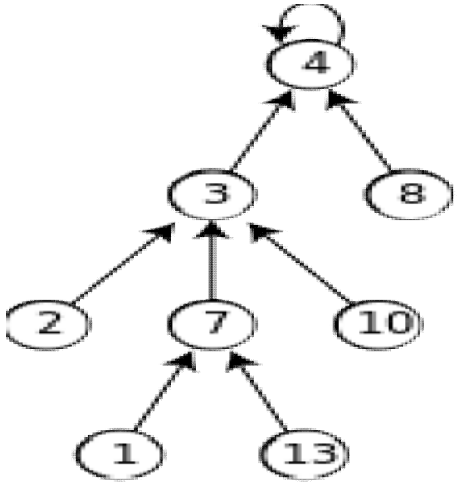
Children $w_1 \ldots w_{d(v)}$ are replaced by arrays $W_1[1 \ldots d(w_1)+1] \ldots W_{d(v)}[1 \ldots d(w_{d(v)})+1]$

Link V[i] to W_i [1] if $i < d(v)+1$

Link W_i [$d(w_i)+1$] to V[i+1]

Each tree of the forest is reduced to a linear list from the first copy of the root to the last one.

Example



From the above parent pointer representation to the following linear list (<u>after doubling the links</u>):

4 → 3 → 2 → 3 → 7 → 1 → 7 → 13 → 7 → 3 → 10 → 3 → 4 → 8 → 4

Applications

Trees in a forest are found by list ranking.

For each list, an array A stores its entries.

For each node, associate a 1 to its first entry (copy) and a -1 to the other ones in a second array B.

Prefix sums on B give in each position the level of the corresponding node in the tree.

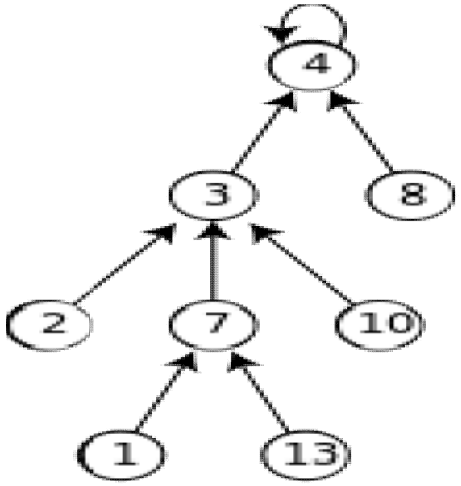
Example

level 1

level 2

level 3

level 4



4 → 3 → 2 → 3 → 7 → 1 → 7 → 13 → 7 → 3 → 10 → 3 → 4 → 8 → 4

4 3 2 3 7 1 7 13 7 3 10 3 4 8 4

1 1 1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1

1 2 3 2 3 4 3 4 3 2 3 2 1 2 1

Computing the Path to an Ancestor

4 3 2 3 7 1 7 13 7 3 10 3 4 8 4

1 2 3 2 3 4 3 4 3 2 3 2 1 2 1

---

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 1 1 1 1 1 1 1 1 1

Prefix sum selects the sublist:

1 7 13 7 3 10 3 4

4 3 4 3 2 3 2 1

Prefix min on the levels finds the path:

4 3 3 3 2 2 2 1

EREW vs CREW PRAM

The Euler tour technique requires $O(\log n)$ time even if $h << n$.

Also, computing each path from a node to its root by the Euler tour technique requires $O(n^2)$ processors even if $h << n$.

If $h << n$, handling cuncurrent reading by standard broadcasting techniques might be more advantageous.

Lempel-Ziv Data Compression

Lempel-Ziv (LZ) coding techniques are based on string factorization.

Zip compressors are based on the so-called LZ1 string factorization process.

On the other hand, LZW compressors are based on the so-called LZ2 string factorization process.

LZ1 Factorization

A finite alphabet, *S* in *A**.

LZ1 factorization: $S = f_1 f_2 \ldots f_i \ldots f_m$ where

- f_i is the longest substring occurring previously in $f_1 f_2 \ldots f_{i-1}$, if f_i is not the empty string;

- f_i is the current alphabet character, otherwise.

Factors are substituted by *pointers* to previous occurences in the input string.

LZ1 Coding

The pointer $q_i=(d_i,l_i)$ codifies f_i, where:

d_i is the displacement back from the position of f_i to the position of its previous occurrence in S if it exists, 0 otherwise;

l_i is the factor length;

If $d_i = 0$, then f_i is an alphabet character and it is inserted uncompressed in the coding of S.

Example

Input String

aabbaabbbabbaa

Factorization

a, a, b, b, aabb, ba, bbaa

Coding

(0,1)*a* (1,1) (0,1)*b* (1,1) (4,4) (5,2) (8,4)

Implementations

In the practical implementations, d_i and l_i are bounded by a constant.

For example, most Zip compressors employ usually a *window* of length 32kB ($d_i \leq 32 \times 10^3$) and 256 is the maximum length of a *match* (factor) ($l_i \leq 256$).

Hashing

Hashing techniques associates a substring in the current position to the position of one of its occurrences in the window (Waterworth [1987], Brent [1987], Whiting, George and Ivey [1991], Gailly and Adler [1991]).

The maximum match length with such position defines the current factor.

Parallelization

The practical implementations of the LZ1 method (zip, gzip, winzip, pkzip) can be executed optimally on a PRAM EREW in logarithmic time by a reduction of the string factorization problem to the problem of computing the path from a leaf to the root in a tree.

Cinque, De Agostino & Lombardi[MCS'10]

Previous Works

Naor [ICALP91]

Crochemore & Rytter [IPL91]

De Agostino & Storer [DCC92]

Nagumo, Lu & Watson [DCC95]

Farach & Muthukrishnan [SPAA95]

Hirschberg and Stauffer [PPL97]

De Agostino [DC00, IS01, PPL04, DCC09]

Reduction: Allocating the Nodes

Allocate $n+1$ nodes $v_1 \ldots v_{n+1}$ where n is the length of the input string S.

v_i is associated with position i in S (v_{n+1} corresponds to the end of the input string).

In parallel for each position i of S, processor i computes the factor matching a copy in one of the positions $i-w, \ldots, i-1$ (w is the window length).

Reduction: Making the Tree

Node v_i is linked with a parent pointer to the node corresponding to the position next to the factor computed in *i.*

We obtain a tree rooted in v_{n+1}.

The factorization of *S* is provided by the path from the leaf v_1 to the root v_{n+1}.

Each node has at most *L* children where *L* is the maximum factor length.

From Parent Pointers to Double Links

F forest of *k*-ary trees with $k=L$.

For each node v_i of *F*, allocate *L* locations initially set to 0.

If node v_j is a child of v_i, then $i - L - 1 < j < i$.

Processor *j* sets the *(i-j)-th* location to 1.

The path from the leaf v_1 to the root v_{n+1} is computed by means of the Euler tour technique.

Parallel Decompression

$q_1 q_2 \ldots q_i \ldots q_m$ is the sequence of pointers encoding the input string S.

$q_i = (d_i, l_i)$ for $1 \leq i \leq m$.

Decompression can be executed optimally on an EREW PRAM in $O(\log n)$ time, where n is the output string length, by reducing such problem to the one of computing the connected components in a forest of rooted trees.

Computing the Factor Positions

$s_0 = 0$; $s_i = l_1 + l_2 + \ldots + l_i$ for $1 \leq i \leq m$

$s_i + 1$ and s_{i+1} are the positions of the first and last character of the $(i+1)$-th factor of the string to compute, for $0 \leq i < m$.

s_m is the string length.

$s_1 \ldots s_m$ are computed by parallel prefix.

Associate to each position i of the string a node v_i, for $1 \leq i \leq s_m$.

The Making of the Forest

The parent array is the following:

- If $d_{i+1} > 0$, for $0 \leq i < m$, positions $s_i+1 \ldots s_{i+1}$ store pointers to positions $s_i+1-d_{i+1} \ldots s_{i+1}-d_{i+1}$, respectively.

- If $d_{i+1} = 0$, then $s_i+1 = s_{i+1}$ is a root and corresponds to the position of an uncompressed character.

Observations

The number of rooted trees in the forest is equal to the alphabet cardinality.

The nodes of each tree correspond to the positions in the string of the alphabet character represented by the root.

Each root corresponds to the first position of a different character in the string.

Each node has at most *w* children.

From Parent Pointers to Double Links

F forest of *k*-ary trees with $k=w$.

For each node v_i of *F*, allocate *w* locations initially set to 0.

If node v_j is a child of v_i, then $i < j < i+w+1$.

Processor *j* sets the *(j-i)-th* location to 1.

The rooted trees of *F* are computed by means of the Euler tour technique.

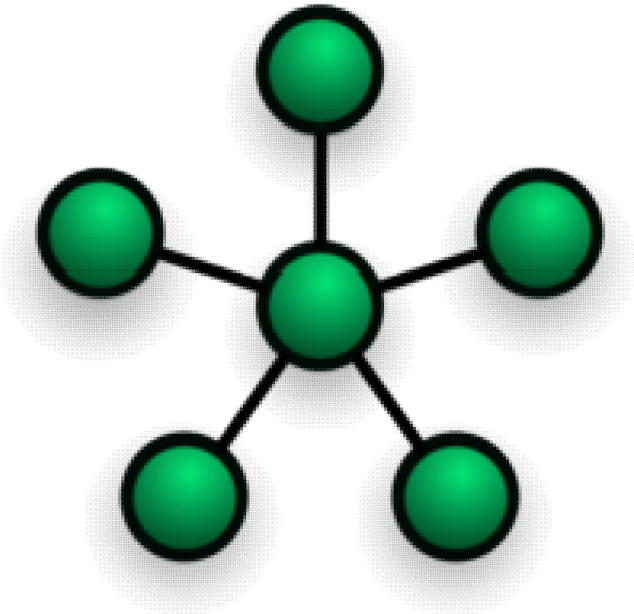
How to Parallelize in Practice

Eliminate or reduce as much as possible the interprocessor communication during the computational phase.

Ensuring scalability and robustness.

An array of processors with distributed memory and no interconnections (except the ones to a central switch) is the simplest model (*star* topology).

From the PRAM to the Star Architecture



Information, exchanged globally among processors in constant time on the PRAM, might take linear time on the Star.

From Global to Local Computation

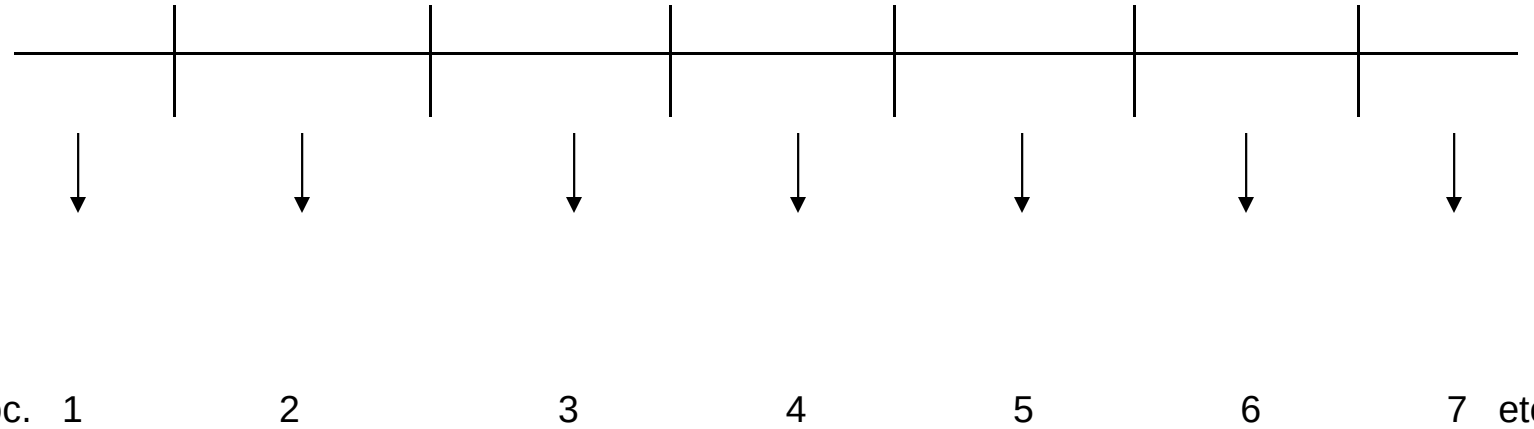
Moving from global to local computation often implies the output of solutions approximating the sequential one.

Scaling up the distributed system usually deteriorates the approximating solution.

This is evident with string factorization if we apply the process in parallel to blocks of data independently.

Approximation

With *n/kw* processors, each processor factorizes a block of length *kw* of *S* with the resulting factorization of the string approximating the sequential one with a multiplicative factor converging to 1 when *k* grows.



Robustness

LZ1 factorization applied to the blocks has the same compression effectivennes of sequential LZ1 factorization for $k \geq 10$.

Since the window length for the the Zip compressors is at least 32kB, the block length should be about 300kB to guarantee robustness.

Scalability

The implementation of a Zip compressor on a large scale distributed system, with no interprocessor communication during the computational phase, is feasible only on large size files (a size about one third of the number of processors in megabytes).

De Agostino [LSD&LAW09]

Cinque, De Agostino & Lombardi [MCS10]

LZ2 Factorization

A finite alphabet, *S* in *A**.

LZ2 factorization: $S = f_1 f_2 \ldots f_i \ldots f_m$ where f_i is the shortest substring which is different from all the previous factors if $i < m$ (differently from LZ1, f_i depends on the previous factors).

The LZW compressor slightly changes the factorization in order not to leave characters uncompressed.

Example

Input String

aabbaabbbabbaab

LZ2 Factorization

a, ab, b, aa, bb, ba, bba, ab

Coding

0a 1b 0b 1a 3b 3a 5a 1b

LZW Factorization and Compression

Finite alphabet *A* with |A| = α, dictionary *D* initially set to *A* with a given order, *S* in *A**.

LZW factorization: $S = f_1 f_2 \ldots f_i \ldots f_m$ where f_i is the longest match with the concatenation of a previous factor and the next character.

LZW compression is obtained by encoding f_i with a pointer to a copy of such concatenation, called *target,* previously added to *D*.

Example

Input String

aabbaabbbabbaab

LZW Factorization

a, a, b, b, aa, bb, ba, bb, aab

Dictionary (initially, equal to *A*)

a, b, aa, ab, bb, ba, aab, bbb, bab, bba

Coding

1 1 2 2 3 5 6 5 7

Observation

$Q = q_1 q_2 \ldots q_i \ldots q_m$ is the sequence of pointers encoding the input string S.

For $1 < i \leq m$, the target of the pointer q_i is the concatenation of the target of the pointer in position q_i - α with the first character of the target of the pointer in position q_i - α + 1, if $q_i > \alpha$.

Parallel Complexity

While Zip and Unzip are in *NC* (the class of problems solvable in polylog time and a polynomial number of processors), LZW is ineherently sequential (*P*-complete).

De Agostino [TCS94]

The *P*-completeness depends only on the factorization process.

The LZW decompressor is parallelizable (asymmetry of the LZW encoder/decoder).

Bounded Memory LZW Factorizations

LZW-FREEZE: f_i is the longest match with one of the first d factors ($d=2^{16}$).

LZW-RESTART: f_i is determined by the last (i mod d) factors.

Standard LZW: RESTART happens when the compression ratio deteriorates.

LZW-LRU: for each f_i , with $i>d$, the *least recently used* (matched) factor is not considered in the next factorization steps.

Parallel Complexity Issues

Sequential FREEZE, RESTART and LRU are parallelizable in theory.

FREEZE is somewhat practical.

RESTART is pseudo-polylog.

Standard LZW is quadratic in the number of processors.

LRU involves huge constant values in its parallel complexity.

The Complexity of LRU

SC^k is the class of problems solvable simultaneously in polynomial time and $O(\log^k n)$ space.

If d is $O(\log^k |S|)$, LRU belongs to SC^{k+1} but it is *hard* for SC^k.

If d is constant, LRU belongs to NC but its parallel complexity involves huge values.

De Agostino and Silvestri [IC2003]

Relaxing LRU (RLRU)

RLRUp partitions the dictionary into *p* classes sorted according to the order of insertion of factors.

An arbitrary factor from the oldest class is deleted at each factorization step.

LZW-RLRU2 is the most efficient Lempel-Ziv compression method (expecially, for highly disseminated heterogeneous data).

LZW and Distributed Computing

As Zip, LZW compression can be applied independently to different blocks of the input string, in parallel.

A block of 600kB garantees robustness (about 300kB to learn a 64K dictionary and about 300kB to compress staticly in order to approximate adaptive RESTART).

Static compression is extremely scalable.

De Agostino [ADVCOMP2014]

Compressing Gigabytes

No static compression is involved with LRU and RLRU.

Parallel processing of at least 600kB blocks is scalable only with large size files.

In such case, LZW-RLRU2 compression guarantees robustness, scalability and a linear speed-up

Standard LZW Compression

Let $Q = Q_1 \ldots Q_i \ldots Q_M$ be the standard LZW encoding of a string S, with Q_i sequence of pointers between two “clear” operations for $1 \leq i \leq M$.

Each Q_i can be decoded independently.

The decoding of Q_i can be parallelized in order to obtain scalability.

Standard LZW Decoding on the PRAM

$q_1 q_2 \ldots\ q_i \ldots q_m$ is a sequence of pointers encoding a substring S' between two consecutive "clear" operations.

For $1 < i \leq m$, 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 $q_i - \alpha + 1$, if $q_i > \alpha$, as for the unbounded version.

Applying Pointer Jumping

Consider $q_1 q_2 \ldots\ q_i \ldots q_m$ as a parent array where:

- if $q_i \leq \alpha$, then position i is a root

- if $q_i > \alpha$, then the parent of i is $q_i - \alpha$

Compute the paths from each node (pointer) to its root (pointer representing an alphabet character).

Parallel Complexity

The maximum target (factor) length L is an upper bound to the height of each tree.

By applying pointer jumping to P, we compute the path from each node to its root in $O(\log L)$ time with $O(n)$ processors on a CREW PRAM, with $n = |S'| \in O(mL)$.

One more step is required to decompress.

Example

output to compute: *aabbaabbbabbbaab*

factorization: *a, a, b, b, aa, bb, ba, bb, aab*

step 0:	1	1	2	2	3	5	6	5	7
step 1:	1	1	2	2	3	5	6	5	7
					1	2	2	2	3
step 3:	1	1	2	2	3	5	6	5	7
					1	2	2	2	3
									1

Last Step

The target of q_i is computed as follows:

the first character is given by the last pointer written on column *i*

the other characters are given by looking at the last pointers on the columns, whose index is equal to one of the other pointers on column i decreased by α – 1

such characters are concatenated according to the bottom-up order of the pointers on column *i*

Work-Optimal Complexity

L equals the dictionary size *d* when the string to compress is unary.

If $d = 2^{16}$, 16 is the theoretical upper bound to the number of iterations.

In practice, $10 < L < 20$, that is, $\log L < 5$.

Ten units are, definitely, a conservative upper bound to the number of global computation steps (each step takes $O(L)$ time if we use only m processors).

The MapReduce Framework

MapReduce (MR) is a framework for processing parallelizable problems across huge datasets using a large number of processors.

Processors could be either on the same local network or shared across geographically and administratively more heterogenous distributed systems.

The MR Programming Paradigm

$P = \mu_1\rho_1 \ldots. \mu_R\rho_R$ is *a* sequence where μ_i is a *mapper* and ρ_i is a *reducer* for $1 \leq i \leq R$.

Mappers broadcast data among the processors.

Reducers execute the computational steps.

The communication cost is determined by the mappers.

Complexity Requirements

- *R* is polylogarithmic in the input size
- sub-linear number of processors
- sub-linear work-space for each processor
- mappers running time is polynomial
- reducers running time is polynomial

Karloff, Suri and Vassilvistkii, A Model of Computation for MapReduce, SODA 2010

MapReduce and CREW PRAM

A CREW PRAM *O(t)* time algorithm using a sub-quadratic number of processors and sub-quadratic global work-space can be implemented in MapReduce in *O(t)* iterations (that is, $R \in O(t)$), satisfying every requirement previously mentioned if *t* is polylogarithmic.

Karloff, Suri and Vassilvistkii, A Model of Computation for MapReduce, SODA 2010

MapReduce and LZW Decompression

$Q = Q_1 \ldots Q_i \ldots Q_M$ with Q_i sequence of pointers between two “clear” operations and $m = \max \{ |Q_i| \}$, for $1 \leq i \leq M$.

The input phase broadcasts the *j-th* pointer of Q_i to processor j, for $1 \leq j \leq m$ and $1 \leq i \leq M$.

The sub-linearity requirements are satisfied since m and M are generally sub-linear in practice.

Stronger Requirements

R is constant (< 10).

Time of a MapReduce operation X the number of processors is *O(T)*, with *T* sequential time (optimality requirement).

The LZW parallel decoder satisfies these stronger requirements.

Paper in preparation

to be submitted

Conclusion

Parallel decompressors are more important than parallel compressors and, generally, have a lower complexity.

In highly distributed implementations, scalability requirements imply a lack of robustness for arbitrary files.

LZW compression has an advantage over Zip since the sequential decoder is highly parallelizable.

Thank you