Parallel Enumeration of *t*–ary Trees in ASC SIMD Model

Zbigniew Kokosiński
zk@pk.edu.pl
Dept. of Automatic Control & Information Technology, Faculty of Electrical & Computer Eng.,
Cracow University of Technology, ul. Warszawska 24, 31-155 Kraków, Poland

Summary
In this paper parallel algorithms are presented for enumeration and unranking of *t*–ary trees with *n* internal nodes. Generation algorithms are designed in the associative computing model ASC that belongs to a broad category of SIMD models. Tree sequences are generated in lexicographical order, with *O(1)* time per object, in a new representation, as combinations with repetitions with restricted growth. The resulting full *t*–ary trees in the form of *z*–sequences and *x*–sequences appear in lexicographical and decreasing lexicographical order, respectively. Sequential *O(n)* ranking and *O(nt)* unranking algorithms for *t*–ary trees with *n* internal nodes are also described on the basis of dynamic programming paradigm. Parallel implementations of ranking and unranking algorithms are discussed. *O(n)* parallel unranking algorithm is derived in the ASC SIMD model.

Key words:
ASC SIMD, t-ary trees, t-sequence, parallel enumeration, parallel generation;

1. Introduction

Combinatorial generation is one of fundamental problems in computer science [1,2]. Combinatorial objects are involved as test or problem instances in numerous important application areas. Many generation algorithms has been developed for such combinatorial objects like *n*–tuples, combinations, permutations, numerical and set partitions, trees, graphs etc.

Binary and *t*–ary trees are important combinatorial structures. Their representation, enumeration and ranking techniques are of great interest from both a theoretical and practical point of view [3,4].
The number of *t*–ary trees with n internal nodes is $B(n,t) = (tn)!/n!(tn-n+1)!$. The number of binary trees $B(n,2)$ is known as the Catalan number $C(n)$. There are well known equivalence relations between such objects as binary trees, full binary trees, ordered trees, well-formed parentheses, standard tableaux and ballot sequences [5].
Many different representations were invented and used for sequential generation of binary and *t*–ary trees, for instance *x*–sequences, (bitstrings), *y*–sequences, *w*–sequences, *z*–sequences, Gray codes etc. [6,7,8,9,10,11,12,13,14,15, 16, 17,18,19,20,21].

Parallel generation algorithms for *t*–ary trees were developed in various SIMD models, providing parallelization of the generation process on the single object level.
The first parallel generation algorithm for the linear array model with N processors ($N = n$) was proposed by Akl and Stojmenović in [22]. In fact one more processor is needed as a master unit what leads to an irregularity of the model with either N or $N+1$ processors. The inversion table representation of t–ary trees is used and subsequent trees are generated with a constant delay. Partial results concerning applications of associative SIMD models with $N+1$ processors for enumeration of *t*–ary trees in the form of *z*–sequences and *x*–sequences were presented in [23,24]. For the above generation methods parallelization of computations on the level of the set of objects is possible by partitioning the full set of objects into *k* disjoint subsets and concurrent subset generation by an adaptive parallel algorithm.
In the paper [25] a parallel adaptive algorithm for generating *t*–ary trees as *z*–sequences in decreasing lexicographical order (A–order) in CREW SM SIMD model was presented, with the processor number N , $N \le n$. The parallelization of computations on the level of single objects was provided, although the object is generated in parts (except the case when $N = n$).
Sequential ranking and unranking algorithms were developed for binary trees [9] and *t*–ary trees [8,14,17, 21,25]. Ranking and unranking of combinatorial configurations is applied in adaptive and random generation algorithms, genetic algorithms, enumeration coding etc. [26,27,28].
One interesting application of generators of combinatorial configurations is associative processing [29,30]. Associative machine models have been shown to have applications in many different areas of parallel computing including processing of data bases, computational geometry, expert systems, artificial intelligence, solving NP–complete and polynomial problems etc. [29,30,31, 32,33,34]. Many efficient algorithms developed in these areas exploit the power of massive associative processing.
Associative processors designed for processing combinatorial problems require additional hardware components, which are able to perform generation of mask/comparand vectors efficiently: a fast interconnection network generating pattern permutations and generators of

combinatorial objects (preferably in bitstring representation). A versatile hardware generator of permutations, combinations and partitions was designed that combines a complex parallel counter as the control unit and a cellular interconnection network [35,36]. Parallel associative algorithms for generation of combinations and partitions in conventional and bitstring representations with *O(1)* time per one generated object [37,38,39] were also designed.

In the present paper a new representation of *t*–ary trees is proposed in the form of *t*–sequences. Then, a new parallel algorithm for the generation of *t*–ary trees is designed. Consecutive objects are generated in a lexicographic order, corresponding to reverse A–order [20], with *O(1)* time per object.

Computations run in the an associative SIMD model called ASC with $n+1$ processors and constant–time basic associative operations [40,41]. In this model parallel processors perform word–parallel bit–serial associative operations.

Sequential algorithms for ranking and unranking *t*–ary trees are also described on the basis of a dynamic programming technique, where tables of pre–computed coefficients are used. The complexity of our new ranking algorithm is only *O(t)* in comparison to *O(nt)* ranking algorithms given in [14,21] and the same as the ranking algorithm in [25]. An *O(nt)* unranking algorithm is also given. Although unranking problems are inherently sequential, a portion of the computations can be parallelized and a new *O(n)* parallel algorithm for unranking *t*–ary trees is constructed in the ASC model. The unranking algorithm given in [25] is $O(nt \log n)$.

The dynamic programming technique was successfully employed in many combinatorial algorithms including unranking combinations [42], partitions [36] and some other combinatorial objects [43,44]. Parallel dynamic programming algorithms were proposed for unranking combinations [45].

Sequential simulation programs for all presented algorithms have been implemented in Pascal and successfully tested for various input data.

The rest of the paper is organized as follows. The next section introduces representations of combinatorial objects. Section 3 describes the ASC SIMD model of computation used throughout this paper. Associative algorithms for generation of *t*–ary trees in two different representations are presented in section 4. Sequential ranking and unranking algorithms for *t*–ary trees together with simple proofs of their correctness are described in section 5. A parallel unranking algorithm is given in section 6. Section 7 contains concluding remarks.

2. Representations and properties of *t*–ary trees

Let us introduce basic notions used throughout this paper and a new representation of *t*–ary trees by means of choice functions of indexed families of sets.

Let $< A_i >_{i \in I}$ denote an *indexed family of sets* $A_i = A$, where: $A = \{1,...,m\}$, $I = \{1,...,n\}$, $1 \le m, n$. Any mapping f which "chooses" one element from each set $A_1,...,A_n$ is called a choice function of the family $< A_i >_{i \in I}$ [46].

With additional restrictions we can model by choice functions various classes of combinatorial objects [35,37,43].

Any choice function $\lambda =< a_i >_{i \in I}$ of the indexed family $< A_i >_{i \in I}$ that satisfies the supplementary condition: $a_i \le a_j$, for $i < j$, and $i, j \in I$, is called a nondecreasing choice function of this family (*l*–sequence). All nondecreasing choice functions are representations of all *n*–subsets with repetitions (combinations with repetitions) of the *m*–element set A. In conventional representation of combinations with repetitions we deal in fact with indexed sets $L_i = \{1,...,m-n+1\} \subset A_i$ [37].

Any choice function $\tau =< a_i >_{i \in I}$ of the indexed family $< A_i >_{i \in I}$ that satisfies the supplementary conditions: 1. $m = (n-1)(t-1)+1$; 2. $a_i \le a_j$, for $i < j$, and $i, j \in I$, and 3. $a_i \in \{1,...,(i-1)(t-1)+1\}$, for $i \in I$, is called nondecreasing choice function with restricted growth of this family (*t*–sequence). In the above mappings we deal in fact with indexed sets $T_i = \{1,...,(i-1)(t-1)+1\} \subset A_i$.

For given n and t, the number of all choice functions τ is the fraction $C_n^{nt}/(nt-n+1)\,C_n^{nt-t+1}$ of the number of all choice functions λ.

There exist bijections between the set of choice functions τ and the set of *t*–ary trees with n internal nodes in other widely used representations. Below we define choice functions ζ and χ corresponding to the notions of *z*–sequences and *x*–sequences known from the literature.

Any choice function $\zeta =< a_i >_{i \in I}$ of the indexed family $< A_i >_{i \in I}$ that satisfies the supplementary conditions: 1. $m = (n-1)t+1$; 2. $a_i \le a_j$, for $i < j$ and $i, j \in I$; and 3. $a_i \in \{1,...,(i-1)t+1\}$, for $i \in I$, is called increasing choice function with restricted growth of this family (*z*–sequence [20]). In the above mappings we deal in fact with indexed sets $Z_i = \{1,...,(i-1)t+1\} \subset A_i$.

Any choice function $\chi =< a_i >_{i \in I}$ of the indexed family $< A_i >_{i \in I}$, where $A_i = X_i = \{0,1\}$ and $I = \{1,...,tn\}$, is

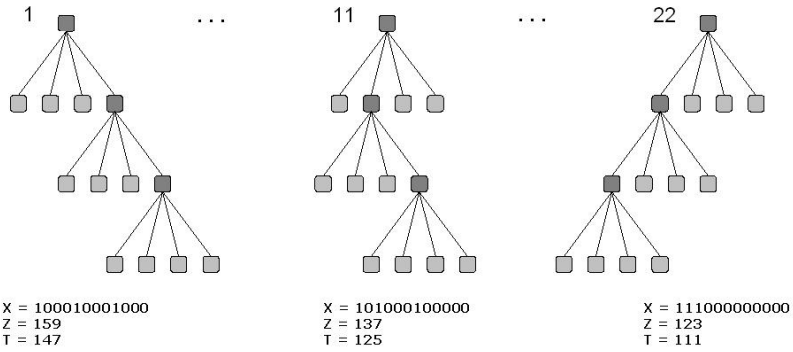

Fig. 1 Representations of *t*-ary trees in A-order; n=3, t=4 and B(3,4)=22.

called binary choice function of this family (*x*–sequence [20]). All binary choice functions such that $a_1+...+a_i \geq i/t$, for $1 \leq i \leq tn$, are bitstring representations of the corresponding *t*–ary trees.

Simple transformations convert choice functions τ into choice functions ζ and χ . For instance, $\zeta[i]=\tau[i]+i-1$, $1 \leq i \leq n$, and $\chi[\zeta[i]]=1$, $1 \leq i \leq n$, and otherwise $\chi[j]=0$, for $1 \leq j \leq nt$.

The *t*–ary trees and their selected representations are depicted in Fig.1.

Let us introduce now lexicographical orders on the set of all choice functions of the family $< A_i >_{i \in I}$.

For given choice functions $\delta =< d_1,...,d_k >$ and $\gamma =< g_1,...,g_k >$, we say that δ is less then γ according to the increasing lexicographical order, if and only if there exists $i, i \in \{1,...,k\}$, satisfying $d_i < g_i$, and $d_j = g_j$, for every $j < i$.

For given choice functions $\delta =< d_1,...,d_k >$ and $\gamma =< g_1,...,g_k >$, we say that δ is less then γ according to the decreasing lexicographical order, if and only if there exists $i, i \in \{1,...,k\}$ satisfying $d_i > g_i$ and $d_j = g_j$, for every $j < i$.

The following proposition is immediate from the previous definitions:

Proposition

Any given choice function τ_p , $1 < p \leq B(n,t)$, may be obtained from the choice function τ_{p-1}, preceding it in the increasing lexicographical order, by incrementing the rightmost element satisfying $\tau_{p-1}[g] < (g-1)(t-1)+1$, and setting all elements $\tau_{p-1}[h]$, $h > g$, to the same value $\tau_{p-1}[g]+1$. □

The above proposition is a validation of the generation algorithm for *t*–sequences and its basic multicast operation as well as the associative model of computation described in section 3.

The number of all *t*–ary trees with *n* internal nodes is denoted by $B(n,t)=(tn)!/n!(tn-n+1)!$. The number of binary trees $B(n,2)$ is known as Catalan number $C(n)$.

Let us introduce now the concept of Ruskey numbers [14], which is used in our dynamic programming ranking/unranking algorithms.

The number of different (n,t) –trees (i.e. trees with *n* internal nodes) less then or equal to $\zeta =< z_1,...,z_{n-i+1}, z_{n-i+2},...,z_n > \in < T_i >_{i \in I}$ with fixed $< z_1,...,z_{n-i+1} >$ (in the increasing lexicographical order) is called a Ruskey number. The Ruskey numbers can be computed recursively by the following formulas:

$$R_n^t(i,j) = j, \tag{1}$$

for $i=1$, and $0 \leq j \leq (n-1)(t-1)+1$;

$$R_n^t(i,j) = R_n^t(i,j-1) + R_n^t(i-1,j), \tag{2}$$

for $1 < i \leq n$, and $(i-1)(t-1)+1 \leq j \leq (n-1)(t-1)+1$.

The above recursive formulas describe construction of Ruskey tables for different values *n* and *t*. Three tables *RT*, containing parts of the Ruskey tables, are shown in Tables 1, 2 and 3.

Ruskey numbers $R_n^t(i,j)$ are stored in corresponding elements $RT[i,j]$ of the table *RT* while all remaining cells are filled with zeros. The following equality holds:

$$R_n^t(i,(i-1)(t-1)+1) = B(n,t), \tag{3}$$

for $1 \leq i \leq n$.

The tables *RT* can be created sequentially in time $O(n^2t)$.

Table 1: Construction of the table RT for (n,2)-trees, n≤9, B(9,2)=4862.

i / j	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	9
2	0	0	2	5	9	14	20	27	35	44
3	0	0	0	5	14	28	48	75	110	154
4	0	0	0	0	14	42	90	165	275	429
5	0	0	0	0	0	42	132	297	572	1001
6	0	0	0	0	0	0	132	429	1001	2002
7	0	0	0	0	0	0	0	429	1430	3432
8	0	0	0	0	0	0	0	0	1430	4862
9	0	0	0	0	0	0	0	0	0	4862

Table 2: Construction of the table RT for (n,3){trees, n≤7, B(7,3)=7752.

i / j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	0	0	0	3	7	12	18	25	33	42	52	63	75	88
3	0	0	0	0	0	12	30	55	88	130	182	245	320	408
4	0	0	0	0	0	0	0	55	143	273	455	700	1020	1428
5	0	0	0	0	0	0	0	0	0	273	728	1428	2448	3876
6	0	0	0	0	0	0	0	0	0	0	0	1428	3876	7752
7	0	0	0	0	0	0	0	0	0	0	0	0	0	7752

Table 3: Construction of the table RT for (n,4){trees, n≤5, B(5,4)=969.

i / j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	0	0	0	0	4	9	15	22	30	39	49	60	72	85
3	0	0	0	0	0	0	0	22	52	91	140	200	272	357
4	0	0	0	0	0	0	0	0	0	0	140	340	612	969
5	0	0	0	0	0	0	0	0	0	0	0	0	0	969

The pre–computed Ruskey tables were used for developing $O(nt)$ ranking and unranking algorithms for t–ary trees (see [14]). The algorithms were developed on the basis of a one–to–one equivalence relation between tree sequences and certain walks in a lattice L constructed for the given values n and t. The dual approach was developed by Zaks [20], where similar tables with coefficients were used.

3. Model of Computation

Parallel computational models significantly differ in versality and complexity [26]. From practical point of view it is reasonable to use dedicated models that provide efficient implementation of key operations necessary for solving the given computational task.

For combinatorial enumeration simple low–level models are sufficient. Let us notice, that generation of the given set of combinatorial objects in a linear order is equivalent to a counting process in the corresponding code. Thus, the most adequate model for parallel generation is the complex parallel counter model, i.e. synchronous parallel counter composed of parallel counters [35,36]. Another simple yet powerful model is the linear array model [47]. Associative SIMD model can also be used for several classes of objects [23,37,38]. In some cases it is possible to combine two simple computational models [39].

Parallel generation algorithms developed later in this paper involve both broadcast and maximum operations. The broadcast (multicast) operation is used for sending data from a source device to all (selected) n destination devices, respectively. The maximum operation is used for finding the maximal element of the given n–element set.

In the popular CREW PRAM model the cost of both above operations is $O(\log n)$. A lower cost of broadcast operation can be obtained in any model of parallel computations with a broadcast bus, where the cost of broadcast/multicast operations is constant. Among many models are PRAM with a broadcast bus, BSR (Broadcasting with Selective Reduction) [26], LARPBS [48], etc. However, a more appropriate model for implementation of our algorithms is a scalable associative processor model with easy–to–implement both constant–time associative searches (relational, maximum/minimum)

and broadcast/multicast operations. Among many models of associative computations, classified as a subclass of SIMD models [49], we selected a well documented model, called ASC [40,41,50].

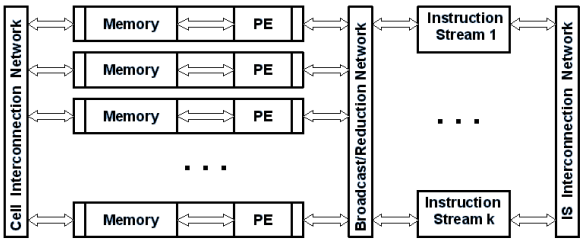

Fig. 2 The ASC model of computation.

Let us describe the most important features of the ASC SIMD model (Fig. 2) that are essential for the presentation of our algorithms (all citations after [41]).

In the ASC model instructions are executed with data–parallelizm, associative searching, maximum and minimum operations are performed in constant–time, synchronization of instruction streams utilize control parallelism. In the simplest ASC model only one instruction stream appears. Such a simple model applied in the present paper contains a number of identical cells and one instruction stream (IS) processor. Each cells consist of a processing element (PE) and a local memory. The memory of an associative computer forms an array of cells. PEs can access the memory in their own cells. Related data items are aggregated together into records and stored one record per cell. More cells then data is available.

Let us now describe the IS processor. The IS processor is connected by a bus with all cells. IS contains a copy of the program being executed and can broadcast an instruction to all cells in unit time. The execution of commands follows SIMD model of parallelizm. An active cell executes the commands it receives from IS. IS can instruct its active cells to perform an associative search. We call successful cells responders, while unsuccessful cells are called nonresponders. It is also possible to restore the former set of active cells. Each above action require one unit of time. IS has also ability to select an arbitrary responder from the set of active cells in unit time and instruct that cell to broadcast its data on the bus. All other cells can receive the value placed on the bus in unit time. The model provides constant time global operations. The IS processor can compute the OR or AND function of a binary value in all active PEs in unit time. Cells with the maximum and minimum value in each of its active PEs can be determined in constant–time.

The ASC SIMD model is of practical value. An FPGA prototype of the ASC processor scalable up to 52 PEs was built using Altera APEX 20K1000E device [51].

In the algorithm *Z–TreeGen* , presented in the next section, the basic operations are associative relational searches $\{\leq,\geq\}$, maximum search and one–to–subset broadcast (multicast) operations. The algorithm requires integer tables $T[n]$, $MAX[n]$, a binary table $TAG[n]$, an integer output table $Z[n]$ and integer variables s and ind.

In the ASC model the algorithm *Z–TreeGen* uses an array built of n cells, each cell containing a record with the cell index i, $1 \leq i \leq n$, integers $T[i]$, $TAG[i]$, $MAX[i]$ and $Z[i]$. The integer variables s and ind are stored in the processor IS. Thus, the required hardware complexity of the model is $O(n)$.

The algorithm *X–TreeGen* uses the same data structures as the algorithm *Z–TreeGen* except the table $Z[n]$. In return for this, the algorithm *X–treeGen* uses the binary output table $X[nt]$ that introduces an irregularity to our model of computation. The size of the output table, determined by the representation of t–ary trees by x–sequences, is t times bigger then the size of the other tables and may cause some problems related to the evaluation of the asymptotic complexity of the computational model. There are at least three solutions to this problem:

(i) Computations related to $X[nt]$ are performed in an augmented ASC model that contains an extra cell array of size nt (controlled by the same IS processor). In this case the total complexity of the augmented ASC model IS $O(nt)$.

(ii) Computations related to $X[nt]$ are performed in an augmented ASC model that contains a special purpose fully parallel associative memory of size nt (controlled by the same IS processor). The extra memory does not need separate PEs for performing associative relational searches. For evaluation purposes the memory of size nt is "partitioned" into n equal parts of size t, each part contributes to the complexity of one cell of the ASC model. Thus, the required hardware complexity of the model remains $O(n)$ (see remark below).

(iii) Computations related to $X[nt]$ are performed in the ASC model in which t additional binary tables $X_j[n]$, $1 \leq j \leq t$, are used for storing the binary output table $X[nt]$. Therefore, each memory record in the ith cell contains the ith bits of all $X_j[n]$ instead of the ith element of $Z[n]$. Thus, the required hardware complexity of the model is $O(n)$ (see remark below).

Remark : The lenght of memory records, assuming nonredundant codings, is $O(logn)$ bits. Therefore, if t is also $O(logn)$ (quite realistic assumption) the total complexity of the augmented ASC model in cases 2 and 3 will remain $O(n)$.

In the next section we assume that either solution 2 or 3 is applied. The parallel algorithm *UnrankTreePar* corresponds to the ASC model with $O(nt)$ processors (see section 6).

4. Parallel Enumeration Algorithms

It is interesting to notice that, despite applying different approaches to the generation task, many enumeration algorithms for given classes of objects reveal a common control structure. For instance, the common control structure of permutation generation algorithms called "factorial counting" was discovered by Sedgewick [52] and this structure was used for the construction of a permutation generator [35].

In this paper we propose that the common control structure for *t*–ary trees with *n* internal nodes in two representations is the structure of (m,n) –combinations with repetitions with restricted growth, where $m=(n-1)(t-1)+1$. The properties of the sequences of combinations with repetitions as nondecreasing choice functions τ are a key factor in our parallelization method (see Proposition 2.1 in section 2). Therefore, the sequence of choice functions τ has been chosen as a basic control sequence for the generation. Actually, other related objects can be obtained from choice functions τ.

For the given input values: n – the number of internal nodes, t – *t*–ary tree parameter, the algorithm *Z–TreeGen* generates in table Z consecutive z –sequences in the increasing lexicographical order. In the algorithm *Z–TreeGen* uniform multicast operations are performed, as described in section 3.

In order to produce control *t*–sequences the algorithm operates on the table *T* and the variable *s*. In the elements of the table *MAX* maximum values of the corresponding *T* elements are stored. In the variable s future values of *T* subsequences are computed and stored in advance. Computations begin with $s=1$. The first *t*–sequence in the table *T* is obtained. Then, the initial table *TAG* is computed and the first output is produced. Next, consecutive values *T* and *s* are produced and output *z*–sequences are computed. The range of table *T* cells used in the procedure *One2subset* is determined associatively in *O(1)* time through two consecutive relational search operations $\{\leq,\geq\}$. The function *Output* performs a parallel conversion of the control *t*–sequence into the output *z*–sequence. Computations run until the last *z*–sequence in the table *Z* is generated, i.e. $ind=1$.

The parallel algorithm *Z–TreeGen* is shown in Fig.3.

```
procedure Z–TreeGen
begin
  ind:=1; s:=1; for i:=1 to n do MAX[i]:=(i—1)(t—1)+1;
  One2subset(s, T, ind, n);
  One2subset(0, TAG, ind, n);
  Output(n, T); ind:=n;
  while ind > 1 do
    if T[ind] < MAX[ind] then
      begin
        s:=T[ind]+1;
        One2subset(s, T, Ind, n);
        if s = MAX[ind] then One2subset(1, TAG, ind, ind);
        if ind < n then One2subset(0, TAG, ind+1, n);
        Output(n, T); ind:=n;
      end;
    else
      ind:= maximum {i: TAG[i]=0};
end;

function One2subset(one, SET, left, right); /multicast/
begin
  for i:=left to right do in parallel SET[i]:=one;
end;

function Output(n, T); /conversion and output/
begin
  for i:=1 to n do in parallel Z[i]:=T[i]+i—1;
  output Z;
end;
```

Fig. 3 The parallel algorithm *Z–TreeGen*.

Sequences generated by the algorithm *Z–TreeGen* , for $n=3$, $t=4$, are listed in Table 4.

In columns 3 and 5 (variable *s* and table *T*) transformations of the control sequence are shown. The bold font points out the **source** and the **destination** elements in multicasts involving *s*, *T* and *TAG*.

Output *z*–sequences are shown in column 6 of Table 4.

Theorem 4.1

Algorithm *Z–TreeGen* generates, in the form of *z*–sequences, all *t*–ary trees with *n* internal nodes in the increasing lexicographical order with constant time per one tree in the associative model ASC with *n+1* processors. Thus, the algorithm *Z–TreeGen* is optimal. □

For the given input values: n – the number of internal nodes, t – *t*–ary tree parameter, the algorithm *X–TreeGen* generates in table *X* consecutive *x*–sequences in the

Table 4: Sequences generated by algorithms *Z-TreeGen* and *X-TreeGen*, for n=3, t=4.

No.	ind	s	TAG	T	Z	X
1	1	1	0 0 0	1 1 1	1 2 3	1 1 1 0 0 0 0 0 0 0 0 0
2	3	2	0 0 0	1 1 2	1 2 4	1 1 0 1 0 0 0 0 0 0 0 0
3	3	3	0 0 0	1 1 3	1 2 5	1 1 0 0 1 0 0 0 0 0 0 0
4	3	4	0 0 0	1 1 4	1 2 6	1 1 0 0 0 1 0 0 0 0 0 0
5	3	5	0 0 0	1 1 5	1 2 7	1 1 0 0 0 0 1 0 0 0 0 0
6	3	6	0 0 0	1 1 6	1 2 8	1 1 0 0 0 0 0 1 0 0 0 0
7	3	7	0 0 1	1 1 7	1 2 9	1 1 0 0 0 0 0 0 1 0 0 0
8	2	2	0 0 0	1 2 2	1 3 4	1 0 1 1 0 0 0 0 0 0 0 0
9	3	3	0 0 0	1 2 3	1 3 5	1 0 1 0 1 0 0 0 0 0 0 0
10	3	4	0 0 0	1 2 4	1 3 6	1 0 1 0 0 1 0 0 0 0 0 0
11	3	5	0 0 0	1 2 5	1 3 7	1 0 1 0 0 0 1 0 0 0 0 0
12	3	6	0 0 0	1 2 6	1 3 8	1 0 1 0 0 0 0 1 0 0 0 0
13	3	7	0 0 1	1 2 7	1 3 9	1 0 1 0 0 0 0 0 1 0 0 0
14	2	3	0 0 0	1 3 3	1 4 5	1 0 0 1 1 0 0 0 0 0 0 0
15	3	4	0 0 0	1 3 4	1 4 6	1 0 0 1 0 1 0 0 0 0 0 0
16	3	5	0 0 0	1 3 5	1 4 7	1 0 0 1 0 0 1 0 0 0 0 0
17	3	6	0 0 0	1 3 6	1 4 8	1 0 0 1 0 0 0 1 0 0 0 0
18	3	7	0 0 1	1 3 7	1 4 9	1 0 0 1 0 0 0 0 1 0 0 0
19	2	4	0 1 0	1 4 4	1 5 6	1 0 0 0 1 1 0 0 0 0 0 0
20	3	4	0 1 0	1 4 5	1 5 7	1 0 0 0 1 0 1 0 0 0 0 0
21	3	4	0 1 0	1 4 6	1 5 8	1 0 0 0 1 0 0 1 0 0 0 0
22	3	4	0 1 1	1 4 7	1 5 9	1 0 0 0 1 0 0 0 1 0 0 0
	1					

decreasing lexicographical order. In the algorithm *X–TreeGen* uniform multicast operations are also essential. In order to produce control *t*–sequences the algorithm operates on elements of the table *T* and the variable *s*. In the elements of the table *MAX* maximum values of the corresponding *T* elements are stored. In variable *S* future values of *T* subsequences are computed and stored in advance. Computations begin with $s=1$. The first *t*–sequence in the table *T* is obtained. Then, the initial table *TAG* and the first output *X* are computed and the first output *X* is produced. Consecutive values *T* and *s* are then produced and output *x*–sequences are computed. Destination cells in tables *T* and *X* used in the procedure *One2subset* are determined associatively in a constant time through two consecutive relational search operations $\{\leq,\geq\}$. In order to produce output *x*–sequences the algorithm operates on elements of the table *X* and binary constants {0,1}. Computations run until the last *x*–sequence in the table *X* is generated, i.e. $ind=1$.

The parallel algorithm *X–TreeGen* for generation of *t*–ary trees is shown in Fig.4.
Output *x*–sequences are shown in column 7 of Table 4.

Theorem 4.2
Algorithm *X–TreeGen* generates, in the form of x–sequences, all t–ary trees with n internal nodes in the decreasing lexicographical order with constant time per one tree in the ASC associative model with n+1 processors. Thus, the algorithm *X–TreeGen* is optimal. □

```
procedure X-TreeGen
begin
   ind:=1; s:=1; for i:=1 to n do MAX[i]:=(i—1)(t—1)+1;
   One2subset(s, T, ind, n);
   One2subset(0, TAG, ind, n);
   One2subset(1, X, ind, n);
   One2subset(0, X, n+1, nt);
   output X; ind:=n;
   while ind > 1 do
      if T[ind] < MAX[ind] then
         begin
            s:=T[ind]+1;
            One2subset(s, T, ind, n);
            One2subset(0, X, T[ind-1]+ind—1, nt);
            One2subset(1, X, T[ind]+ind—1, T[ind]+n—1);
            if s = MAX[ind] then One2subset(1, TAG, ind, ind);
            if ind < n then One2subset(0, TAG, ind+1, n);
            output X;
            ind:=n;
         end
      else ind:= maximum {i: TAG[i]=0};;;
end;

function One2subset(one, SET, left, right); /multicast/
begin
   for i:=left to right do in parallel SET[i]:=one;
end;
```

Fig. 4 The parallel algorithm *X–TreeGen*.

5. Ranking and Unranking Algorithms

In this section we assume *t*–ary trees to be represented by increasing choice fuctions with restricted growth (*z*–sequences). In the algorithms *UnrankTree* and *RankTree* presented below the table *RT* is used, which includes a part of the Ruskey table. Each coefficient $R_n^t(i, j)$ is mapped to the cell $RT[i, j]$. In the dynamic programming approach the table *RT* with Ruskey numbers is pre–computed.

For the given input values: *n* – the number of internal nodes, *t*– *t*–ary tree parameter, *Index* – rank of the *z*–sequence in the increasing lexicographical order ($1 \le Index \le B(n,t)$), *RT* – table with elements $RT[i, j]$ containing the Ruskey numbers $R_n^t(i, j)$, the algorithm *UnrankTree* produces the corresponding *z*–sequence in table *Z*.

Computations proceed with tree rank $Index'$ in the decreasing lexicographical order. In order to determine *Z* the table *RT* is searched. The maximum elements $RT[i,m]$, satisfying the given inequality $RT[i, j] \le Index'$, are selected in each row. Then, the next value $Z[n-i+1]$ is computed. After *O(nt)* iterations we obtain the required *z*–sequence in the table *Z*.

```
procedure UnrankTree
begin
  i:=n; j:=(t—1)(n—1);
  Index':=RT[i, j+1]—Index;
  while (Index' ≥ 0) and (i >0) do
    if RT[i, j] ≤ Index' then
      begin
        Index':=Index'—RT[i, j];
        Z[n—i+1]:=(tn—t+2)—(i+j);
         i:=i—1;
      end
    else j:=j—1;;;
  return Z;
end;
```

Fig. 5 The algorithm *UnrankTree*.

Example 1

For the input data set {A, B, C} given below compute table Z using the algorithm *UnrankTree*.

Input A

n=9, t=2 and Index(Z)=3682.

Solution

i=9, j=8, Index'=RT[9,9]—Index=4862—3682=1180.
RT[9,8]=0 ≤ Index'=1180, Index'=1180—0=1180, Z[1]:=1.
RT[8,7]=0 ≤ Index'=1180, Index'=1180—0=1180, Z[2]:=3.
RT[7,7]=429 ≤ Index'=1180, Index'=1180—429=751, Z[3]:=4.
RT[6,7]=429 ≤ Index'=751, Index'=751—429=322, Z[4]:=5.
RT[5,7]=297 ≤ Index'=322, Index'=322—297=25, Z[5]:=6.
RT[4,4]=14 ≤ Index'=25, Index'=25—14=11, Z[6]:=10.
RT[3,3]=5 ≤ Index'=11, Index'=11—5=6, Z[7]:=12.
RT[2,3]=5 ≤ Index'=6, Index'=6—5=1, Z[8]:=13.
RT[1,1]=1 ≤ Index'=1, Index'=1—1=0, Z[9]:=16.

Input B

n=7, t=3 and Index(Z)=6409.

Solution

i=7, j=13, Index'=RT[9,9]—Index=7752—6409=1343.
Z=[1,4,5,6,8,11,14].

Input C

n=5, t=4 and Index(Z)=425.

Solution

i=5, j=13, Index'=RT[5,13]—Index=969—425=544.
Z[1,3,4,12,17].

Theorem 5.1

Algorithm *UnrankTree* is correct and its asymptotic complexity is $O(nt)$.

Proof. The set of all $B(n,t)$ trees can be displayed in the form of a rooted ordered tree of height n (see Fig. 6).

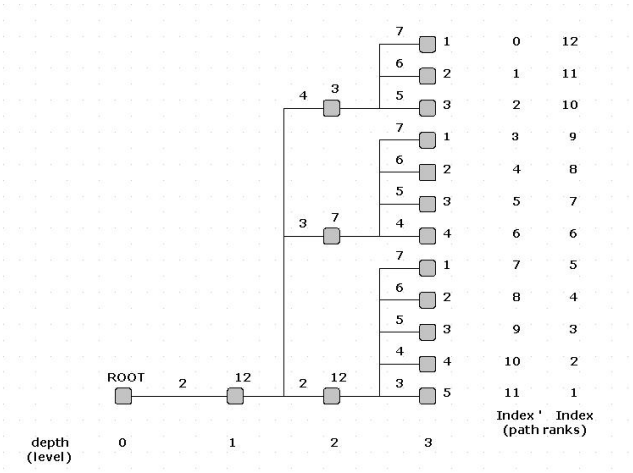

Fig. 6 The rooted ordered tree of all B(3,3) trees with edge and node labels.

There are $n(t-1)-1$ nodes with depth *n*. Each node with depth *i*, $0 \le i \le n-1$, has $it-k+1$ children, where *k* is an integer label for the edge connecting the given node with its ancestor (for the root that has no ancestor we assume $k=0$), and edges connecting the given node with its descendants are labeled by $k+1, k+2, \ldots, it+1$, respectively. In this way all nodes with depth *i* as well as all paths are ordered in the tree.

Traversing the tree in preorder and listing all paths from the root to subsequent leaves – by sequences of edge labels – is equivalent to generation (enumeration) of all $B(n,t)$ trees in increasing lexicographic order. Let us assign to all such paths their ranks in decreasing lexicographic order. Unranking the object with rank *Index* in the tree is equivalent to finding in the tree the path with rank $Index' = B(n,t) - Index$, $1 \le Index \le B(n,t)$.

Every node of the tree with depth *i* has an integer label equal to the sum of all leaves of ordered subtree rooted in

this node and all its siblings with depth *i* following it. Each node label is a Ruskey coefficient. We determine the path with rank $Index'$ by determining a proper subtrees on the consequtive levels starting from the root. Rooted subtrees on the ith level are viewed in the decreasing order of their size (size means in this case the number of subtree leaves). In order to do this the current $Index'$ of the choice function is compared with node labels $R_n^t(i, j)$ and taken from the cell $RT[i, j]$. In each level i no more then $(n-i+1)(t-1)-1$ comparisons are made and before the next step rank $Index'$ is modified. Single iteration with complexity *O(1)* is repeated *O(nt)* times. Condition $RT[i, j] \le Index'$ is satisfied *n* times, and the next item of the required object is obtained. Hence, the total complexity of the algorithm is *O(nt)*. □

For the sake of completeness the *O(n)* ranking algorithm *RankTree* is presented that has lower asymptotic complexity then the *O(nt)* ranking algorithms described by Ruskey [14] and Zaks [21].

For the given input values: n – the number of internal nodes, t – t–ary tree parameter, Z – table with z–sequence, RT — table with elements RT[i, j] containing the Ruskey numbers $R_n^t(i, j)$, the algorithm *RankTree* computes $Index$, i.e. the rank of the z–sequence in the increasing lexicographical order, $1 \le Index \le B(n,t)$.

Computations proceed with tree rank $Index'$ in the decreasing lexicographical order. The value of $Index'$ is updated iteratively. After *O(n)* iterations we obtain the rank $Index'$ which is then converted into the tree rank $Index$ in the increasing lexicographical order.

```
procedure RankTree
begin
   Index':=0;
   m:=(t—1)(n—1);
   for j:=1 to n do Index':=Index'+RT[n—j+1, m—Z[j]+j];
   Index:=RT[n-1, m+1]—Index';
   return Index.
end;
```

Fig. 7 The algorithm *RankTree*.

Example 2

For the input data set {D, E, F} given below find *index(Z)* using the algorithm *RankTree* .

Input D

n=9, t=2 and Z=[1,3,4,5,6,10,12,13,16].

Solution

Index'= RT[9,8]+RT[8,7]+RT[7,7]+RT[6,7]+RT[5,7]+RT[4,4]+
+RT[3,3]+RT[2,3]+RT[1,1]=0+0+429+429+297+14+5+5+1=
=1180.

Index=RT[9,9]—Index'=4862—1180=250.

Input E

n=7, t=3 and Z=[1,4,5,6,8,11,14].

Solution

Index'= RT[7,12]+RT[6,10]+RT[5,10]+RT[4,10]+RT[3,9]+
+RT[2,7]+RT[1,5]=0+0+728+455+130+25+5=1343.

Index=RT[7,13]—Index'=7752—1343=6409.

Input F

n=5, t=4 and Z=[1,3,4,12,17].

Solution

Index'= RT[5,12]+RT[4,11]+RT[3,11]+RT[2,4]+RT[1,0]=
=0+340+200+4+0=544.

Index=RT4[5,13]—Index'=969—544=425.

Theorem 5.2

Algorithm *RankTree* is correct ind its asymptotic complexity is *O(n)*.

Proof. Correctness of the ranking method results directly from the proof of Theorem 5.1 and the original paper by Ruskey [14]. The complexity of the algorithm is obviously O(n). □

6. Parallel Ranking and Unranking Algorithms

In the algorithm *UnrankTree* two computational processes can be parallelized – creation of the coefficient table *RT*, and searching in the rows of the coefficient table *RT*. The parallelization may be achieved with the help of a special purpose model.

Let us notice that elements in the *i*th row of the table *RT* form a sequence which is increasing with column index j. This property is essential for speeding up the search in the *RT* rows. For a given pair (n,t), sequential generation of the table *RT* requires $O(n^2t)$ steps. Generation of the table *RT* from recursive formulas presented in section 2 may be parallelized through systolic computations and the generation time may be reduced from $O(n^2t)$ to *O(nt)* .

The algorithm *UnrankTreePar* has the same input and output as in the algorithm *UnrankTree*.

Computations proceed with tree ranks in the decreasing lexicographical order. In order to determine table Z associative searches are used. In each iteration of the **for** loop the element $RT[i,k]$ with maximum kth coordinate is selected satisfying the given inequality $\{\le\}$ and the value of $Index$ variable is updated. Then the next value $Z[n-i+1]$ is obtained and the value m is updated. After n iterations we obtain the corresponding z–sequence in the table Z.

A simple parallel unranking algorithm for t –ary trees implementing associative relational search operation $\{\le\}$ and maximum operation is given in Fig. 8.

```
procedure UnrankTreePar
begin
  m:=(n—1)(t—1);
  Index':=RT[n, m+1]—Index;
  for i:=n downto 1 do
    begin
      k:= maximum {j: (0 ≤ j ≤ m) and (RT[i, j] ≤ Index');
      Index':=Index'—RT[i, k];
      Z[n—i+1]:=(tn—t+2)—(i+j);
      m:=k;
    end;
  return Z;
end;
```

Fig. 8 The algorithm *UnrankTreePar*.

The unranking algorithm *UnrankTreePar* is a variant of algorithm *UnrankTree* .

Theorem 6.1
Algorithm *UnrankTreePar* is correct and its asymptotic complexity is *O(n)*.

Proof. Correctness of the method results from the Proof of Theorem 5.1. Search in consecutive rows of the coefficient table *RT* is organized in an associative manner. Rank $Index'$ is simultaneously compared with all values stored in the cells of the *i*th row of the table *RT*. This reduces the search time in the *i*th row to *O(1)*. In this way the value $Z[n-i+1]$ is determined. Before the next step the rank $Index'$ is modified. Each iteration has time complexity *O(1)*. The number of iterations is *n*, hence the total complexity of the algorithm is *O(n)*. □

In order to execute the algorithm *UnrankTreePar* in the ASC model of computation, where associative computations are performed in the cell array, it is necessary to apply $(n-1)(t-1)+2$ cells, each cell containing one *n*–element column of the table *RT*. Thus, in the algorithm the indices *i* and *j* must be mutually exchanged. In this case, the total complexity of the ASC model is *O(nt)*.

In the algorithm *RankTree* computations in **for** loop can be parallelized. CRCW PRAM model with concurrent write of the sum of elements $RT[n-j+1, m-Z[j]+j]$ to the result variable $Index'$ may be applied. The resulting parallel ranking algorithm has *O(1)* asymptotic time complexity. The algorithm is obvious and it is omitted here.

7. Concluding Remarks

Associative algorithms *Z–TreeGen* and *X–TreeGen* for generation of *t*–ary trees with *n* internal nodes provide the parallelization of computations on the level of single combinatorial object, satisfying most properties discussed in [22,47]. They can be used in adaptive tree generation too, enabling further parallelization on the set of objects level. In this case standard unranking techniques for *t*–ary trees may be applied with a little effort for programming a number of generators working in parallel.
Two unranking algorithms have also been presented in Zaks' representation. They are derived on the basis of the dynamic programming paradigm and apply the coefficient table *RT* containing Ruskey numbers. The parallel unranking algorithm *UnrankTreePar* can be performed in a special purpose parallel processor containing both systolic and associative features. At first, the coefficient table *RT* is created by systolic computations in *O(nt)* time. Then, *n* subsequent elements of a tree codeword (*z*–sequence) are computed in *O(1)* time per element through bit–parallel word–parallel associative relational searches and maximum operations.
The sequential *O(n)* algorithm *RankTree* is as fast as similar algorithms developed earlier. A parallel version of this algorithm in CRCW PRAM model requires only a constant time.
The ASC SIMD model of computation was selected mostly due to its versatility. Readers familiar with principles of associative computing may notice that the bit–serial word–parallel paradigm represented by the ASC model can be replaced by the bit–parallel word–parallel paradigm [53,54] which was used in [23]. The asymptotic time complexities of the related algorithms for a fully parallel associative processor remain the same as their counterparts in ASC model.

References

[1] Knuth, D.E. (2004) The art of computer programming. Pre–fascicles of chapter 7.2, (Addison–Wesley).
[2] Ruskey, F. (2003) Combinatorial generation, Working Version (1j-CSC 425/520), electronic publication available at http://www.edu/~ruskey/book.pdf
[3] Knuth, D.E. (1997) The art of computer programming. Fundamental algorithms. (Addison-Wesley).
[4] Knuth, D.E. (2006) The art of computer programming. Fascile 4. Generating all trees – History of Combinatorial Generation, (Addison-Wesley).
[5] Stanton, D., White, D. (1996) Constructive combinatorics. (Wiley–Interscience).
[6] Ahrabian, H., Nowzari–Dalini, A., Salehi, E. (2004) Gray code algorithm for listing k-ary trees, *Studies in Informatics and Control*, 243–251.

[7] Er, M.C. (1987) Lexicographic listing and ranking t–ary trees, *The Computer Journal*, 559–572.
[8] Er, M.C. (1992) Efficient generation of k–ary trees in natural order, *The Computer Journal*, 306–308.
[9] Knott, G.D. (1977) A numbering system for binary trees, *Comm. ACM*, , 113–115.
[10] Korsh, J.F. (1994) Loopless generation of k–ary tree sequences. *Information Processing Letters*, 243–247.
[11] Mäkinen, E. (1991) A survey of binary tree codings. *The Computer Journal*, 438–443.
[12] Roelants van Baronaigien, D. (1991) A loopless algorithm for generating binary tree sequences. *Information Processing Letters*, 189–194.
[13] Roelants van Baronaigien, D., Ruskey, F. (1988) Generating t–ary trees in a–order. *Information Processing Letters*, 205–213.
[14] Ruskey, F. (1978) Generating t–ary trees lexicographically, *SIAM Journal of Computing*, 424–439.
[15] Skarbek, W. (1988) Generating ordered trees, *Theoretical Computer Science*, 153–159.
[16] Skarbek, W. (2007) On generating all binary trees, *Fundamenta Informaticae*, 505–536.
[17] Trojanowski, A.E. (1978) Ranking and listing algorithms for k–ary trees. *SIAM Journal of Computing*, 492–509.
[18] Vajnovszki, V. (1996) Constant time algorithm for generating tree Gray codes, *Studies in Informatics and Control*, 15–21.
[19] Xiang L., Ushijima K., Akl S.G. (2000) Generating regular k–ary trees efficiently, *The Computer Journal*, 290–300.
[20] Zaks, S. (1980) Lexicographic generation of ordered trees, *Theoretical Computer Science*, 63–82.
[21] Zaks, S. (1982) Generating and ranking t–ary trees, *Information Processing Letters*, 44–48.
[22] Akl, S.G., Stojmenović, I. (1996) Generating t–ary trees in parallel, *Nordic J. of Computing*, 63–71.
[23] Kokosiński, Z. (2002) On parallel generation of t–ary trees in an associative model. In: Proc. PPAM'2001, *Lecture Notes in Computer Science*, 228–235.
[24] Kokosiński, Z. (2004) A parallel dynamic programming algorithm for unranking t–ary trees. In: Proc. PPAM'2003, *Lecture Notes in Computer Science*, 255–260.
[25] Ahrabian, H., Nowzari–Dalini, A. (2007) Parallel generation of t–ary trees in A–order, *The Computer Journal*, 581-588.
[26] Akl, S.G. (1997) Parallel computation: models and methods. (Prentice Hall), 475–509.
[27] Üçoluk, G. (1996) A method for chromosome handling of r–permutations of n–element set in genetic algorithms. *Proc. 1996 IEEE Int. Conference on Evolutionary Computation*, Indianapolis, USA, 55–58.
[28] Tomic R.V. (2005) Quantized Indexing: Beyond Arithmetic Coding, *1stWorks Corporation Technical Report TR05-0625*, 32 pp. (available at http://www.1stWorks.com)
[29] Kapralski, A. (1994) Sequential and parallel processing in depth search machines. (World Scientific).
[30] Krikelis, A., Weems, C.C. (Eds) (1997) Associative processing and processors. (IEEE Computer Society Press).
[31] Kapralski, A. (1992) Supercomputing for solving a class of NP–complete and isomorphic complete problems, *Computer Systems Science and Engineering*, 218–228.
[32] Kokosiński, Z. (1997) An associative processor for multi–comparand parallel searching and its selected applications. *Proc. 3rd Int. Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, Vol.III, 1434–1442.
[33] Nepomniaschaya A.S., Dvoskina M.A. (2000) A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors, *Fundamenta Informaticae*, 227–243.
[34] Nepomniaschaya A.S., Kokosiński, Z. (2004) Associative graph processor and its properties, *Proc. Int. Conference on Parallel Computing in Electrical Engineering*, Dresden, Germany, IEEE Computer Society, 297–302.
[35] Kokosiński, Z. (1990) On generation of permutations through decomposition of symmetric groups into cosets. *BIT*, 583–591.
[36] Kokosiński, Z. (1993) Circuits generating combinatorial configurations for sequential and parallel systems. *Monograph 160*, Politechnika Krakowska, Kraków, Poland (in Polish), 106 pp.
[37] Kokosiński, Z. (1997) On parallel generation of combinations in associative processor architectures. *Proc. Int. Conference on Parallel and Distributed Systems*, Barcelona, Spain, 283–289.
[38] Kokosiński, Z. (1999) On parallel generation of set partitions in associative processor architectures. *Proc. 5th Int. Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, Vol.III, 1257–1262.
[39] Kokosiński, Z. (2006) A new algorithm for generation of exactly m–block set partitions in associative model. In: Proc. PPAM'2005, *Lecture Notes in Computer Science*, 67–74.
[40] Potter, J.L. (1992) Associative computing. A programming paradigm for massively parallel computers. (Plenum Press).
[41] Potter, J.L., Baker, J.W., and *all* (1994), ASC: an associative computing paradigm. *Computer*, 19–25.
[42] Kokosiński, Z. (1995) Algorithms for unranking combinations and their applications. *Proc. 7th IASTED Int. Conf. Parallel and Distributed Computing Systems*, Washington D.C., USA, 216–224.
[43] Kapralski, A. (1993) New methods for the generation of permutations, combinations and other combinatorial objects in parallel, *Journal of Parallel and Distributed Computing*, 315–326.
[44] Kapralski, A. (2000) Modelling arbitrary sets of combinatorial objects and their sequential and parallel generation. *Studia Informatica*, Silesian University of Technology (a monograph)
[45] Kokosiński, Z. (1996) Unranking combinations in parallel. *Proc. 2nd Int. Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, USA, Vol.I, 79–82.
[46] Mirsky, L. (1971) Transversal theory. (Academic Press).
[47] Akl, S.G., Stojmenović, I. (1996) Generating combinatorial objects on a linear array of processors. In: Zomaya, A.Y., (Ed): Parallel computing; paradigms and applications. (Int. Thompson Comp. Press) 639–670.
[48] Pan, Y., Li, K. (1998) Linear array with a reconfigurable pipelined bus system - concepts and applications. *Journal of Information Sciences*, 237–258.

[49] Parhami, B. (1999) Introduction to parallel processing: algorithms and architectures. (Plenum Press).
[50] ASC_Research_Papers, http://www.cs.kent.edu/~potter/research/papers/
[51] Wang, H., Walker R.A. (2003) Implementing a scalable ASC processor, *Proc. 17th Int. Parallel and Distributed Processing Symposium, Workshop in Massively Parallel Processing* Nice, France, 7 pp., electronic publication available at http://doi.ieeecomputersociety.org/10.1109/IPDPS.2003.1213482
[52] Sedgewick, R. (1977) Permutation generation methods. *Computing Survey*, 137–164.
[53] Foster, C.C. (1976) Content addressable parallel processors. (Van Nostrand Reinhold).
[54] Yau, S.S., Fung, H.S. (1977) Associative processor architecture – a survey, *Computing Survey*, 3–27.

Zbigniew Kokosiński received his M.S. degree in 1982 from Cracow University of Technology, Kraków, Poland. In 1992 he received Ph.D. degree with distinction in Computer Science from the Gdańsk University of Technology, Gdańsk, Poland. In 1994-1997 he was employed as an Assistant Professor at the Department of Computer Software, University of Aizu, Aizu-Wakamatsu, Japan. Currently, dr. Kokosiński is an Assistant Professor at the Dept. of Automatic Control and Information Technology, Faculty of Electrical and Computer Engineering, Cracow University of Technology, Kraków. His research is focused on combinatorial optimization and parallel metaheuristics, generation of combinatorial objects in parallel, associative processors and algorithms, programmable devices and systems. The publications include over 40 refereed papers in international scientific journals and conference proceedings. Reviews for J. of Parallel and Distributed Computing, Networks, Parallel Computing, Lecture Notes in Computer Science, J. of Mathematical Modeling and Algorithms, Kragujevac J. of Mathematics etc. Dr. Kokosinski participated in organization of several international conferences in the area of parallel computing serving as a referee, program committee member, session chair etc. (PDCS, ICEC, IPPS, PDPTA, PPAM, FPL, PARELEC, MCCSIS, ISPDC). For many years he was a member of the IEEE Computer Society, ACM and IASTED. Biographical notes: Marquis "Who's Who in Science and Engineering".