STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 2, 2000

PARALLEL VERIFICATION AND ENUMERATION OF TOURNAMENTS

GÁBOR PÉCSY AND LÁSZLÓ SZŰCS

ABSTRACT. The area of tournaments is extensively discussed in literature. In this article the authors introduce asymptotically optimal sequential algorithms for the verification of score vectors and score sequences and a sequential polynomial algorithm for enumeration of complete tournaments. The extensions of these algorithms to different parallel architectures including CREW PRAM, linear array, mesh and hypercube are also presented. It is shown that most of the parallel algorithms discussed here are work-optimal extensions of the sequential ones.

1. INTRODUCTION

Round-robin tournaments are popular in the world of sport, games or elections and they are very much discussed in computer science as well. A tournament is an $n \times n$ real matrix. The elements of the main diagonal t_{ii} equal to zero and the pairs of symmetric elements $t_{ij} : t_{ji}$ give the result of the match between P_i (the i-th player) and P_j. The sum of the elements of the i-th row (s_i) is called the score of the i-th player. A non-decreasingly ordered sequence of the scores is the score sequence of the tournament.

The most usually discussed problems regarding tournaments include:

- Verification of a score sequence/score vector means the decision if there exists a tournament for a given score sequence/score vector.
- Enumeration of score sequences means the counting of the possible different score sequences for a given number of players (n).

The outline of the paper is as follows. The following section describes the problems and the used computational models more formally. Section 3 deals with verification problems and their sequential and parallel solutions.Then Section 4

2000 *Mathematics Subject Classification.* 05C20, 68Q25, 65Y05.

1998 *CR Categories and Descriptors.* G.2.1 [**Discrete mathematics**]: Combinatorics – *Counting problems*; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems – *Computations on discrete structures* C.1.4 [**Processor architectures**]: Parallel architectures – *Distributed architectures* .

presents our results about enumeration. Finally, a table summarizes the results with possible future works.

2. Basic notions

2.1. Tournaments.

Tournament: A round-robin tournament is an $n \times n$ real matrix $T_n = [t_{ij}]$ $(n \geq 2)$. The elements of the main diagonal t_{ii} equal to zero and the pairs of symmetric elements $t_{ij} : t_{ji}$ give the result of the match between P_i (the i-th player) and P_j. $t_{ij} = t_{ji}$ means a draw, while $t_{ij} > t_{ji}$ means the win of P_i against P_j.

Score vector: The sum of the elements of the i-th row (s_i) is called the score of the i-th player and the vector $(s_1, \ldots, s_n)$ is called the score vector of the tournament.

Score sequence: A non-decreasingly ordered sequence of the scores is denoted by $q =< q_1, \ldots, q_n >$ and is called the score sequence of the tournament.

Complete tournament: We call a tournament complete if in it the permitted elements are 0 and 1 and the sum of the symmetric elements $(t_{ij}+t_{ji}$, where $i \neq j)$ is always 1. A set of tournaments is called complete for a given n if it contains all possible n player complete tournaments.

2.2. Computational models.

Sequential model: A RAM running pseudo-code similar to structured programming languages.

PRAM: Parallel RAM, consists of a shared memory and possibly infinite number of RAMs which operate with the same pseudo code as in the sequential case. Depending on the methods of accessing the shared memory there are different types of PRAM.

EREW: Exclusive Read Exclusive Write
ERCW: Exclusive Read Concurrent Write
CREW: Concurrent Read Exclusive Write
CRCW: Concurrent Read Concurrent Write

As concurrent read of shared memory is usually allowed while the result of concurrent write is ambigous we decided to use CREW PRAM in our study.

Linear array: A linear array consists of p processors (named $1, 2, \ldots, p$). Processor i has two direct bidirectional interconnection links to its neighbouring processors ($i-1$ and $i+1$) except processor 1 and p which has only one neighbour.

Mesh: A mesh is an $a \times b$ grid in which there is a processor at each grid point. The edges correspond to communication links and are bidirectional. In this paper we consider only square meshes, where $a = b$.

Hypercube: A hypercube of dimension d has $p = 2^d$ processors. Each processor can be labeled with a d-bit binary number. A processor is connected only to those processors which label differs in only one bit.

Work-optimal: We call a parallel algorithm work-optimal compared to a given sequential algorithm if $\frac{P_n * p}{S_n} = O(1)$, where S_n is the run time of the sequential algorithm, P_n is the run time of the parallel algorithm and p is the number of processors.

Notice that if a parallel algorithm is work-optimal compared to a given asymptotically optimal sequential algorithm then the parallel algorithm itself is asymptotically optimal as well.

3. Verification

Verification of a score sequence/score vector means the decision if there exists a tournament for a given score sequence/score vector. Landau [5] proved the following theorem which gives necessary and sufficient condition for the existence of a complete tournament for a particular score sequence.

Theorem 1. *A non-decreasing sequence of* n *integers* $< q_1, \ldots, q_n >$ *is a score sequence if and only if*

$$\sum_{i=1}^{k} q_i \geq \binom{k}{2}$$

for each $k = 1, 2, \ldots, n$ *with equality for* $k = n$.

3.1. Sequential algorithm.

Theorem 1 can be directly applied to verify score sequences as they are ordered non-decreasingly. The following algorithm solves this problem in $\Theta(n)$ time and with $O(1)$ auxiliary memory.

```
1  s:=0; i:=1; ok:=(q_n<n);
2  while i<n and ok loop
3      s:=s+q_i;
4      ok:=s≥(i*(i-1)/2);
5      i:=i+1;
6  end loop
7  ok:=ok and (s+q_n)=(n*(n-1)/2);
8  return ok;
```

Algorithm 1: Sequential algorithm for score sequence verification

As the trivial lower bound for the verification problem is $\Omega(n)$ — the algorithm has to read the input — Algorithm 1 is asymptotically optimal for score sequence verification.

In case of score vectors the input is not ordered properly so Theorem 1 (and Algorithm 1) can not be applied directly. One possible solution is to sort the input and then apply Algorithm 1 to the result. It is known that sorting of general keys takes $\Omega(n \log n)$ time but if keys are integer numbers from the range [0..k] then they can be sorted in $O(max(n, k))$ time. Such algorithm can be found in chapter 9 of [1]. In case of a score vector all elements must belong to range [0..n-1] so the vector can be sorted in $O(n)$ time. This condition can also be verified in $O(n)$ time, so we get the following algorithm.

Step 1: Verify whether all elements in the vector fall in the range [0..n-1]. If not then the input can not be a score vector.
Step 2: Sort the input.
Step 3: Use Algorithm 1.

Algorithm 2: Sequential verification of score vectors

Note that Algorithm 2 is asymptotically optimal for the same reason as Algorithm 1.

3.2. **Parallel algorithms.** In this section we provide an efficient way to implement Algorithm 1 and Algorithm 2 on different parallel architectures. [1]

3.2.1. *PRAM.* On a CREW PRAM Algorithm 1 can be implemented in a very straightforward way.

Step 1: For all processors compute the prefix-sums (r_i) of the input sequence.
Step 2: Processor p_i $(i := 1, 2, \ldots, n-1)$ calculates $l_i := (r_i \geq (i * (i-1)/2))$ while p_n calculates $l_n := (r_n = (n * (n-1)/2))$.
Step 3: Calculate $OK := l_1 \wedge \ldots \wedge l_n$ using the prefix computation algorithm with all processors.

Algorithm 3: Simple parallel algorithm for score sequence verification

Step 1 can be done in $O(\log n)$ time, Step 2 takes $O(1)$ time and Step 3 is $O(\log n)$ again. Note that in case of CRCW PRAM Step 3 takes only $O(1)$ time.

This algorithm uses $O(n)$ processors and operates in $O(\log n)$ time therefore it is not work-optimal, but it can be improved to run on $O(\frac{n}{\log n})$ processors in $O(\log n)$ time which is work-optimal. To achieve this we divide the input

[1]It is assumed that the reader is familiar with the prefix-sum computation and other well-known parallel algorithms summarized in [2] as they are building blocks of the following algorithms.

into $\log n$ long pieces. Processor p_i will sequentially calculate prefix-sums of $s_{(i-1)\log n+1}, \ldots, s_{i \log n}$. Then the processors will apply the original prefix computation algorithm on the sums of the pieces. In the third step each processor will update the prefix-sums of the corresponding piece by adding the sum of all the previous pieces. After this the processors will calculate l_i values sequentially for all elements belonging to them and finally they perform a prefix computation using the same algorithm as for the prefix-sum calculation to determine $OK := l_1 \wedge \ldots \wedge l_n$.

Step 1: For all processors compute sequentially the prefix-sums ($t_{i,j}$, where $i := 1, 2, \ldots, \frac{n}{\log n}; j := 1, 2, \ldots, \log n$) of the corresponding piece of input sequence.
Step 2: For all processors compute the prefix-sums ($r_{i \log n}$) of $t_{i,\log n}$.
Step 3: For all processors compute sequentially $r_{(i-1)\log n+j} := r_{(i-1)\log n} + t_{i,j}$ $(i := 1, 2, \ldots, \frac{n}{\log n}; j := 1, 2, \ldots, \log n)$.
Step 4: Processor p_i $(i := 1, 2, \ldots, \log n)$ calculates $l_{(i-1)\log n+j} := (r_{(i-1)\log n+j} \geq (((i-1)\log n + j) * ((i-1) * \log n + j - 1)/2))$ using equality at the last position.
Step 5: Calculate $OK := l_1 \wedge \ldots \wedge l_n$ using the prefix computation algorithm described in Step 1–3, with all processors.

Algorithm 4: Work-optimal verification of score sequences on CREW PRAM

In this algorithm all steps take $O(\log n)$ time so the whole algorithm works in $O(\log n)$ time as well. It uses $O(\frac{n}{\log n})$ processors so this is a work-optimal parallelization of Algorithm 1. As Algorithm 1 is asymptotically optimal algorithm for the score sequence verification problem the same holds for Algorithm 4 as well.

Notice that in this algorithm only the parallel steps (Step 2 and 5) require interprocessor communication and these steps are all parts of prefix computations.

3.2.2. *Linear array.* A lower bound on every interconnection networks for a problem is the diameter of the network if all processors of the network contributes to the computation of the result. As the diameter of a linear array of n processors is $n-1$, $\Omega(n)$ is a lower bound for the score sequence and score vector verification problems. These problems can be solved in $O(n)$ time on a single processor as well, the trivial (and optimal) solution is to send all data to the first processor of the array – this can be done in $O(n)$ time – and do the verification there, using the sequential algorithms. These solution are work-optimal if and only if the number of processors in the array is $O(1)$.

3.2.3. *Mesh.* The diameter of a p processor mesh is $\sqrt{p}$, so $\Omega(\sqrt{p})$ is a lower bound to an algorithm. The mesh adaptation of Algorithm 3 solves the problem in $O(\sqrt{n})$ if $p = n$. But we can apply the same technique as in Algorithm 4. If we assign $n^{\frac{1}{3}}$ element for each processor of a $n^{\frac{1}{3}} \times n^{\frac{1}{3}}$ mesh then both the sequential and the

parallel steps work in $O(n^{\frac{1}{3}})$ time. The number of processors in this case is $n^{\frac{2}{3}}$ so the algorithm is work-optimal.

3.2.4. *Hypercube.* In a p processor hypercube, prefix computation can be performed in $O(\log p)$ time, which means that adoptations of Algorithm 3 and Algorithm 4 can work in the same time bounds as in case of CREW PRAM.

3.2.5. *Parallel score vector verification.* Unfortunately there is no known work-optimal parallel sorting algorithm for integer key from a given domain. This means that the most difficult step in the parallel adoptation of Algorithm 2 is the sorting. The complexity of sorting usually exceeds the complexity of the other steps so the overall complexity of the algorithm equals the complexity of sorting the input. For PRAM and hypercube there are algorithms which can sort general keys in $O(\log^2 n)$ time.

4. Enumeration

Enumeration of score sequences means the counting of the possible different score sequences for a given number of players (n).

For $n > 1$ let $f_n(T, E)$ be the number of non-decreasing sequences of integers satisfying

$$\sum_{i=1}^{n} q_i = T, q_n = E \text{ and } \sum_{i=1}^{k} q_i \geq \binom{k}{2}, k = 1, 2, \ldots, n-1.$$

Narayana and Bent in [7] presented a recursive formula for determining $f_n(T, E)$:

$$f_1(T,E) = \begin{cases} 1, & \text{if } T = E \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

$$\text{for } n \geq 2 \qquad (1)$$

$$f_n(T,E) = \begin{cases} \sum_{k=0}^{E} f_{n-1}(T-E, k), & \text{if } T - E \geq \binom{k}{2} \\ 0, & \text{otherwise.} \end{cases}$$

Let t_n be the number of possible score sequences in case of n players. For $n > 1$ we have the following formula for t_n:

$$t_n = \sum_{E=r}^{n-1} f_n(\tbinom{n}{2}, E), \text{where } r = \left\lfloor \frac{n}{2} \right\rfloor.$$

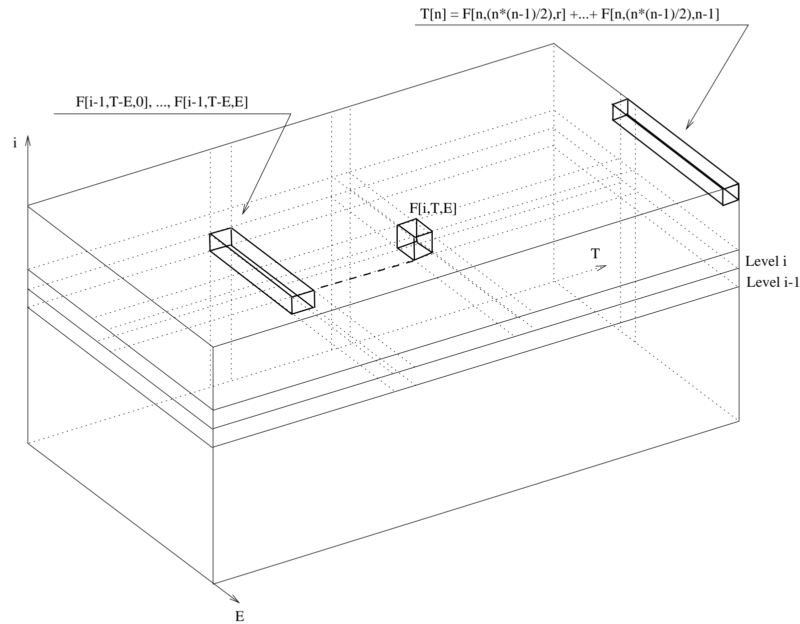

FIGURE 1. Array element dependencies in the non-optimized formula

4.1. **Units of measure.** The experimental results indicate that the value of t_n is increasing exponentially with n ($t_n = \Omega(2^n)$) which implies that we need $\log t_n = \Theta(n)$ memory to store a single number. This also implies that addition of such numbers takes $\Theta(n)$. In the next parts of the article we will count the number of operations (addition, send, receive and assignment) on the elements of the array during the analysis of the algorithms. In a real implementation all of these operations can be done in $O(n)$ time.

4.2. **Sequential algorithms.** The most straightforward recursive calculation of t_n using the recursive formula (1) has exponential run time so it is not applicable for bigger n values. Using dynamic programming the run time can be reduced significantly into polynomial domain.

4.2.1. *Algorithm using dynamic programming.* The following algorithm uses array of size $n \times n \times (n(n-1)/2 + 1)$ elements and works in $\Theta(n^5)$ time.

The operation of the algorithms can be divided into two phases. First phase is filling in the array F which contains the values of $f_i(T, E)$ for $i = 1..n, T = 0..\frac{n(n-1)}{2}$ and $E = 0..n-1$, thus the dimensions of the array are $n \times \frac{n(n-1)}{2} + 1 \times n = \Theta(n^4)$. Calculating a particular $F[i, T, E]$ element takes $\Theta(1)$ time for $i = 1$ —

```
1  for i:=1 to n loop
2      for T:=0 to (n*(n-1)/2) loop
3          for E:=0 to n-1 loop
4              if i=1 then
5                  if T=E then
6                      F[i,T,E]:=1;
7                  else
8                      F[i,T,E]:=0;
9                  end if;
10             else
11                 F[i,T,E]:=0;
12                 if (T-E)≥ ((i-1)*(i-2)/2) then
13                     for k:=0 to E loop
14                         F[i,T,E]:=F[i,T,E]+F[i-1,T-E,k];
15                     end loop;
16                 end if;
17             end if;
18         end loop;
19     end loop;
20 end loop;
21 TN:=0;
22 for E:=(n div 2) to n-1 loop
23     TN:=TN+F[n,(n*(n-1)/2,E];
24 end loop;
25 return TN;
```

Algorithm 5: Calculating the number of score sequences using dynamic programming

lines 5–9 — and $O(n)$ otherwise — lines 11–16 (see Figure 1). This means that the whole algorithm runs in $O(n^5)$ time. The second phase is to calculate the number of score sequences (TN) using the filled array F (see Figure 1).

4.2.2. *Improved algorithm.* In Algorithm 5 the number of the used array elements is $\Theta(n^4)$ so $O(n^4)$ is a lower bound to the run time of any solution using this approach, but the run time is $O(n^5)$. We show that using a proper reformulation of equation (1) the run time of the algorithms can be reduced to $\Theta(n^4)$.

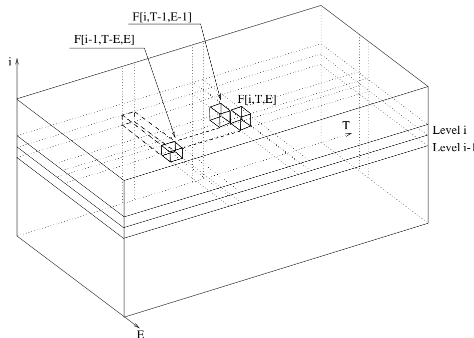

FIGURE 2. Dependencies of elements in the array in case of the optimized formula

$$
\begin{aligned}
f_i(T,E) &= \sum_{k=0}^{E} f_{i-1}(T-E,k) \\
&= \sum_{k=0}^{E-1} f_{i-1}((T-1)-(E-1),k) + f_{i-1}(T-E,E) \\
&= f_i(T-1,E-1) + f_{i-1}(T-E,E)
\end{aligned}
\tag{2}
$$

Notice that when we compute $f_i(T,E)$ we already know $f_i(T-1,E-1)$ so we can replace the loop in lines 13–15 of Algorithm 5 with a simple assignment (see Figure 2).

4.3. **Parallel algorithms.**

4.3.1. *PRAM.* A straightforward parallel implementation of the non-optimized formula is the following. We fill in one level of the array in one round. We have $\frac{n}{\log n}$ processors for each element in the level (figure 3). These processors perform a prefix computation to calculate the value using the original formula (1). This takes $O(\log n)$ time. The array has n levels so the whole algorithm runs in $O(n \log n)$. On a single level of the array there are $n\binom{n}{2}$ elements, which means that we need $n\binom{n}{2}\frac{n}{\log n} = \frac{n^3(n-1)}{2\log n} = O(\frac{n^4}{\log n})$ processors to achieve this. Unfortunately this solution is not work-optimal as the amount of work done is $O(n^4 \log n) * O(n \log n) = O(n^5)$.

This algorithm used the property of (1) that the value of a particular element in a certain level depends on other elements from a lower level only. This way we could avoid the synchronization overhead between the processors working on different elements. Using the optimized formula we have to use results from the same level

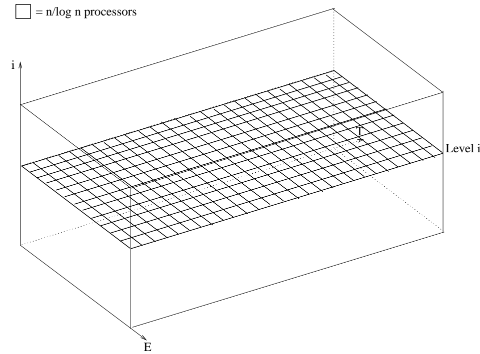

FIGURE 3. Using brute-force approach on PRAM architecture

as well. More accurately the value of $f_i(T, E)$ depends on $f_i(T-1, E-1)$ which in turn depends on $f_i(T-2, E-2)$ etc. This dependency limits the maximum number of processors that a work-optimal algorithm can utilize.

Here we present three possible work-optimal algorithms, using n, n^2 and $\frac{n^3-n}{2\lceil\log n\rceil}$ processors.

Each algorithm calculates the values level by level. The first version assigns a processor to each possible values of T and these processors calculate $f_i(T, E)$ for the different E values one by one. As the value of T belongs to the domain $[0..\binom{n}{2}]$ so we need $\binom{n}{2}+1$ processors and each calculates $f_i(T, E)$ for $E = 0, \ldots, n-1$ which requires $\Theta(n)$ time. There are n levels in the array so the whole run time of the algorithm is $\Theta(n^2)$.

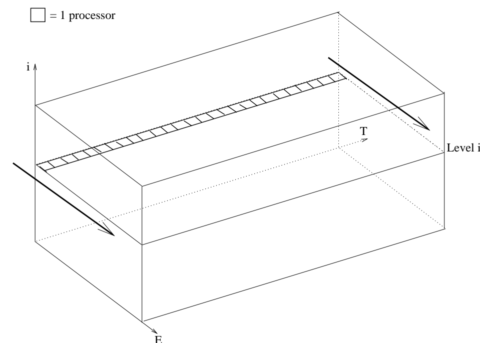

FIGURE 4. PRAM algorithm using n^2 processors

The second algorithm assigns processors to each possible values of E and these processors calculate $f_i(T, E)$ for the different T values one by one. This means

that we need n processors and due to symmetry the run time of this algorithm is $\Theta(n^3)$.

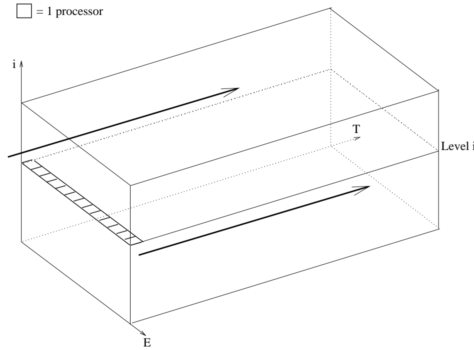

FIGURE 5. PRAM algorithm using n processors

The third algorithm uses a bit different approach. For this algorithm, computation of one level takes two steps. During the first step the processors set the elements of the level to 0. There are $\frac{n^3-n}{2}$ elements in a level, we have $\frac{n^3-n}{2\lceil \log n \rceil}$ processors so it takes $O(\log n)$ time to accomplish. In the second step the algorithm calculates $f_i(T+j, E+j), (j = 1..n)$ using prefix computation algorithm with $\frac{n}{\log n}$ processors on $f_{i-1}(T-E, j)$. This also takes $\log n$ time, so a single level can be calculated in $\log n$ time, the array has n levels so the whole algorithm works in $O(n \log n)$.

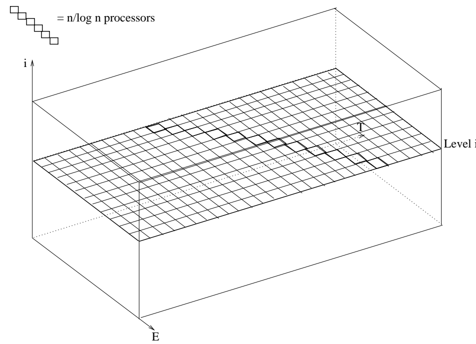

FIGURE 6. PRAM algorithm using $\frac{n^3}{\log n}$ processors

4.3.2. *Linear array.* The second work-optimal algorithm given for PRAM can be adapted to n processor linear array as well. Each processor is assigned a possible value of E. The processor stores the two-dimensional subarray belonging to that particular value . The processors use Algorithm 6.

Step 1: For i:=1 each processor calculates $F[i, T, E] := (T = E)?1 : 0$.
Step 2: For i:=2..n each processor performs Algorithm 7.
Step 3: The processors perform a prefix computation to determine t_n.

Algorithm 6: Enumeration of score sequences on n processor linear array

```
Each processor (E:=0..n-1) on level i (i:=2..n) does the following:
1  for T:=1 to n*(n-1)/2 loop
2      if E > 0 and T > 0) then
3          receive Z:=F[i,T-1,E-1] from processor E-1;
4      else
5          Z:=(i=2 and T=0 and E=0)?1:0;
6      end if;
7      if T-E ≥ ((i-1)*(i-2)/2) then
8          F[i,T,E]:=Z+F[i-1,T-E,E];
9      else
10         F[i,T,E]:=0;
11     end if;
12     if E < n-1 and T < n*(n-1)/2 then
13         send F[i,T,E] to processor E+1;
14     end if;
15 end loop;
```

Algorithm 7: Calculating $f_i(T, E)$ values on an n processor linear array

4.3.3. *Mesh.* As linear array can be embedded to a mesh the algorithm given in the previous section can be applied for meshes as well.

However there exists another work-optimal algorithm using n^2 processors. Let the processors be indexed from 1 to n (i) and from 0 to $n-1$ (E). Processor (i, E) has a one-dimensional subarray containing $f_i(T, E)$ for the possible different T values. This way to calculate $f_i(T, E)$ for a particular value of T it has to communicate with two of its neighbours.

The enumeration problem can be solved using the following algorithms:

4.3.4. *Hypercube.* As mesh can be embedded to a hypercube the same algorithms given for meshes can be applied.

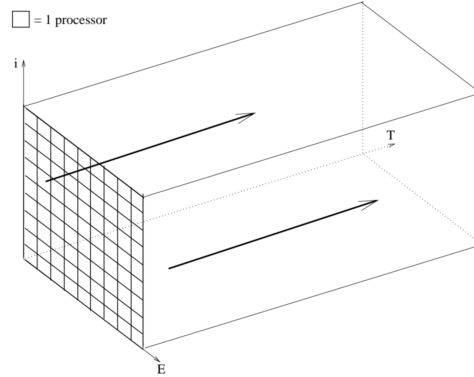

FIGURE 7. Mesh algorithm using n^2 processors

Step 1: For i:=1 and E:=0..n-1 each processor performs Algorithm 9. For i:=2..n and E:=0..n-1 each processor performs Algorithm 10.
Step 2: The processors $(n, 0), \ldots, (n, n-1)$ perform a prefix computation to determine t_n.
Algorithm 8: Enumeration of score sequences on n^2 processor mesh

5. Further optimizations

The algorithms given in the previous section use $\Theta(n^4)$ array elements. It's easy to see that each algorithm (except the last one) at a given time uses only two levels of the array. Calculating the ith level we need the (i-1)th one for that. This means that we don't have to store all levels only the current and the previous one. This optimization will reduce the number of necessary elements to $2 * n * n * (n-1)/2 = n^3 - n^2 = \Theta(n^3)$.

6. Conclusions

6.1. **Summary.** The table below summarizes our results for p processors and n-player tournaments:

Problem	**Sequential**	**Linear array**	**Mesh**	**Hypercube**	**PRAM**
Score sequence	$\Theta(n)$	$\forall p \in \mathbb{N}$ $\Theta(n)$	$p = n^{\frac{2}{3}}$ $\Theta(n^{\frac{1}{3}})$ work-opt.	$p = \frac{n}{\log n}$ $\Theta(\log n)$ work-opt.	$p = \frac{n}{\log n}$ $\Theta(\log n)$ work-opt.

Score vector	$\Theta(n)$	$\forall p \in \mathbb{N}$ $\Theta(n)$	$p = n$ $O(n^{\frac{1}{2}})$	$p = n$ $O(\log^2 n)$	$p = n$ $O(\log^2 n)$
				$p = n^2$ $O(\log n)$	$p = n^2$ $O(\log n)$
Enume-ration of score sequences	Recursive formula with dynamic programming: $\Theta(n^4)$	$p = n$ $\Theta(n^3)$ work-opt.	$p = O(n)$ $\Theta(n^3)$ work-opt.	$p = O(n)$ $\Theta(n^3)$ work-opt.	$p = n$ $\Theta(n^3)$ work-opt.
			$p = n^2$ $\Theta(n^2)$ work-opt.	$p = O(n^2)$ $\Theta(n^2)$ work-opt.	$p = n^2$ $\Theta(n^2)$ work-opt.
					$p = \frac{n^3-n}{2\lfloor \log n \rfloor}$ $\Theta(n \log n)$ work-opt.

The notion of completeness of tournaments can be extended to k-completeness.

k-complete: We call a tournament k-complete if its elements are non-negative integers and the sum of the symmetric elements is always k $(t_{ij} + t_{ji} = k, \text{where } i \neq j)$. A set of tournaments is called k-complete for a given n if it contains all possible n player k-complete tournaments.

From the definition it follows that a complete tournament is 1-complete. The theorems and algorithms presented above can be easily extended to k-complete tournaments.

6.2. **Future Works.** In this section we try to identify some possible directions to do further research.

- Fine-tuning the presented non work-optimal algorithms if possible or design new ones.
- As we saw the value of t_n increases exponentially this also implies that $f_i(T, E)$ values are increasing in similar order. Storing such values requires $O(n)$ bits. However it is possible that the average size of the elements in the array is smaller.
- The task of reconstruction means that for a given score sequence we construct a tournament. The asimptotically optimal sequential algorithms solve this problem in $\Theta(n^2)$ time. Parallel reconstructing algorithms for the problem are to be considered.
- Parallel algorithm for calculating the lexicographical successor of a given score sequence.
- Parallel listing of score sequences for a given n.

Each processor (i,E) (i:=2..n; E:=0..n-1) does the following:

```
1  for T:=0 to n*(n-1)/2 loop
2      if E>0 and T>0 then
3          receive Z:=F[i,T-1,E-1] from processor (i,E-1);
4      else
5          Z:=(i=2 and T=0 and E=0)?1:0;
6      end if;
7      if T-E≥((i-1)*(i-2)/2) then
8          if T=0 then
9              Y:=(i=2 and E=0)?1:0;
10         else
11             receive Y:=F[i-1,T-E,E] from processor (i-1,E);
12         end if;
13         F[i,T,E]:=Z+Y;
14     else
15         F[i,T,E]:=0;
16     end if;
17     if T<n*(n-1)/2 then
18         if E<n-1 then
19             send F[i,T,E] to processor (i,E+1);
20         end if;
21         send F[i,T-E,E] to processor (i+1,E);
22     end if;
23 end loop;
```

Algorithm 9: Calculating $f_i(T,E)$ values for $i > 2$

Each processor (1,E) (E:=0..n-1) does the following:

```
1  for T:=0 to n*(n-1)/2 loop
2      F[1,T,E]:=(E=T)?1:0;
3      if T<n*(n-1)/2 and T≥E then
4          send F[1,T-E,E] to processor (2,E);
5      end if;
6  end loop;
```

Algorithm 10: Calculating $f_1(T,E)$ values

The techniques that were used in the presented algorithms aimed the parallel adoption of a sequential dynamic programming solution. These techniques should be extended to other algorithms using dynamic programming.

Acknowledgement. The authors would like to thank Antal Iványi for sharing his knowledge about tournaments and being open to discuss our ideas.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest (1990), *Introduction to Algorithms*, McGraw-Hill, MIT Press, New York.

[2] E. Horowitz, S. Sahni, S. Rajasekaran (1998), *Computer Algorithms*, Computer Science Press, New York.

[3] A. Iványi, *Good tournaments*, submitted to Annales Univ. Sci. Budapest., Sectio Math.

[4] A. Iványi, *Maximal tournaments*, In: Fourth Join Conf. on Math. and Comp. Sci. Felix, June 5–10, 2001, submitted to Pure Math. and Appl.

[5] H. G. Landau (1953), *The condition for a score structure III*, Bull. Math. Biophysics, pp. 153–158.

[6] J. W. Moon (1968), *Topics on Tournaments*, Holt, Rinehart & Winston, New York.

[7] T. V. Narayana, D. H. Bent (1964), *Computation of the number of score sequences in round-robin tournaments*, Canad. Math. Bull. 7 (1), pp. 133–136.

[8] K. B. Reid (1996), *Tournaments: scores, kings, generalizations and special topics*, Congressus Numerantium 115, pp. 171–211.

Department of General Computer Science, Eötvös Loránd University, 1117 Budapest, Pázmány Péter sétány 1/B., Hungary

E-mail address: `pici@elte.hu` and `slice@elte.hu`