Short Note

Generating Permutations on a VLSI Suitable Linear Network

A parallel algorithm for generating all the $k!$ permutations of ${}^kP_k$ for every $(1 \leqslant k \leqslant n)$ is presented. The architecture consists of a linear processor array with n elements. The kth processor receives a permutation p of ${}^{k-1}P_{k-1}$ from the $(k-1)$th processor and intercalates k at all the k possible positions of the sequence p, one at a time. After each intercalation it sends the permutation obtained to the $(k+1)$th processor and also outputs it. In this way the nth processor outputs all the $n!$ permutations of ${}^nP_n$ in $(n+n!)$ units of time and our profit are all the ${}^kP_k$ $(1 \leqslant k < N)$ which output is included in that time. The network is VLSI implementable and fault tolerant. It is shown how to find the position of a given permutation and how to obtain the permutation of a given position, where position refers to the generation order of the permutations by each processor. With a simple modification in the algorithm performed by the processors, the network is able to generate combinations.

Received December 1987, revised January 1989

1. Introduction

Generating permutations and combinations of objects are important problems in combinatorics. Several algorithms have been proposed in order to solve them in different ways. Mor and Fraenkel[8] and Zaks[10] studied how to generate all the $n!$ permutations of n elements (${}^nP_n$). Gupta and Bhattacharjee proposed an algorithm for finding all the ${}^nP_r$, the distinct permutations of r items out of n objects.[4] Semba showed how to get all k-subsets $(1 \leqslant k \leqslant n)$ of the set $\{1, 2, \ldots, n\}$ in lexicographic order[9] and Chen and Chern generated the permutations of at most k out of n objects.[1]

When dealing with permutations we are always concerned by either the amount of time or the amount of memory space we take to generate them. As ${}^nP_n = n!$, the product $A \cdot T$ (where T stands for Time and A for Area) gives intractable values when n grows. Our aim in this paper is to present a permutations generator that could answer some of the usual requests in this area: *little memory space; very simple basic operations; modularity; local control; fault tolerance; VLSI suitability* and *no extra work*.

Our processor network generates all the permutations of ${}^kP_k$ for every $(1 \leqslant k \leqslant n)$. It is based on the fact that if we have $p = b_1 b_2 \ldots b_{k-1}$ — a ${}^{k-1}P_{k-1}$ permutation —, then we can obtain k permutations of ${}^kP_k$ by intercalating k at all the k possible positions of the sequence p, one at a time. So, we can generate all the permutations of ${}^kP_k$ from ${}^{k-1}P_{k-1}$ by executing the very same operation on each element of ${}^{k-1}P_{k-1}$.

The processor network is composed by n simple processors computing the permutations in parallel so that the kth processor is able to output the $k!$ distinct permutations of k elements in $(k+k!)$ units of time, for $k = 1, 2, \ldots, n$. Moreover the outputs are pipelined since the network is synchronized by the slowest processor behaviour. Therefore the whole process requires $(n+n!)$ units of time to generate all the $\Sigma k!$ permutations.

The network architecture is a linear array composed of processors and connections between neighbours. The global behaviour is of MIMD (Multiple Instruction Multiple Data) type and no global synchronization is necessary. Moreover each processor holds a private memory and no shared storage is available so that communicating is done by message passing.[6]

The main characteristics of our solution are that it needs no global control, the kth processor should store only a vector of dimension $(k+1)$ and the basic operation performed inside a processor is quite so simple as transposing two adjacent elements of its vector.

The paper organization is as follows: in section 2 we introduce our solution and in section 3 we discuss it regarding the seven aims stated above. In section 4 it is shown explicitly the relation between the permutations and their order of generation, while in section 5 we show how to use the same network to generate combinations and present some experimental results obtained with the FPS T-20 4-Cube parallel computer.[5]

2. The problem and the proposed solution

There are several techniques for generating permutations of a set. For instance, Mor and Fraenkel generate them in parallel 'by performing the same transformation on previous blocks of permutations'.[8] Zaks constructs them sequentially and a new one is obtained 'by reversing a certain suffix of its predecessor'.[10] Chen and Chern propose a ring of processors controlled by a host, each one with a stack; the operations are 'a cyclic right shift of the objects held in all the top elements of the stacks' and a subsequent test on the stack in order to discover if a new permutation has been generated.[1]

Instead of generating the permutations of k elements we are going to generate all the $\Sigma k!$ permutations of ${}^kP_k$ for $1 \leqslant k \leqslant n$, with n processors, in $O(n!)$ total time and linear space per processor. We start with ${}^1P_1 = 1$. Each kth processor $(1 < k < n)$ reads a permutation $p = b_1 b_2 \ldots b_{k-1}$ from the $(k-1)$th processor, intercalates k at all possible positions of the sequence p, one at a time, and sends it to the $(k+1)$th processor as well as it outputs the permutation. The nth processor does the same operations except the sendings, for it has no right neighbour.

The network is synchronized by the data flow. When a processor finishes its work it sends a token ($PERM[1] = 0$) toward its right neighbour. In order to well initialize and synchronize the whole process, processor PE_1 sends ${}^1P_1 = \{1\}$ and then $PERM[1] = 0$. After receiving the token, each processor retransmits it to its right neighbour and stops computation. When $k = n$ there are no sendings to the right, for the nth processor is the last one.

The algorithm performed inside the kth processor is as follows (written in an OCCAM-like language):[7]

Algorithm 1:

```
Receive (left, PERM[1:k])
  {read a permutation from the left neighbour}
WHILE PERM[1] <> 0
  {there is something to read}
  SEQ
  PERM[k+1]: = k
  SEQi = [0 FOR k]
  SEQ
  PERM[k-i+1] <=> PERM[k-i]
  {exchange two adjacent el's}
  PAR
    Send(right, PERM[1::k+1])
    {send PERM to the right}
    Send(bottom, PERM[1::k])
    {output PERM}
    Receive(left, PERM[1::k])
Send(right, PERM[1::k+1])
{send PERM[1] = 0 to the right neighbour}
```

Let the time unit be the time spent by the nth processor in performing one transposition operation and one I/O operation. Reading, sending and outputting can be performed in parallel. However, since the communication time depends on the length of vector $PERM$, the time unit is not exactly the same for all the processors. Let f be the time taken by the first permutation to reach the nth processor. Then it follows by induction that (cf^3):

Proposition 1:

The kth processor constructs all the ${}^kP_k$ permutations, $1 \leqslant k \leqslant n$.

Proposition 2:

The network generates all the $\Sigma k!$ permutations $(1 \leqslant k \leqslant n)$ in $(f+n!)$ units of time.

3. Back to our aims

Each processor performs the same simple set of instructions, so we can say that the model of computation is a SPMD (Single Program Multiple Data) with no global control. However, the algorithm we propose is globally asynchronous. Local synchronization corresponds to the communications between neighbour processors and can hence be obtained through a send/receive protocol: PE_k can send a message to PE_j if and only if PE_k and PE_j are neighbours and PE_j is ready to receive the message. Thus, no control is needed for synchronization.

We adopted this send/receive protocol because of the characteristics of our network. A kth processor works k units of time with a permutation p_i read from its predecessor. At the same time the $(k-1)$th processor worked only one unit of time with a permutation read

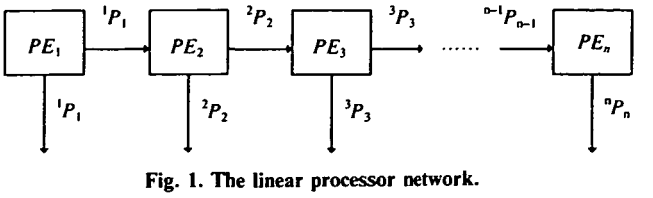

Fig. 1. The linear processor network.

from its predecessor. When processor k goes reading p_{i+1} from processor $(k-1)$, this one is already ready for sending it. So we lose no time implementing such a kind of protocol.

At the Introduction we stated seven major goals we had when conceiving this permutations generator. We will then discuss how the network matches them.

The kth processor needs only $k+1$ memory cells to hold vector *PERM*, hence a *little memory space* is required per processor. Besides, a *very simple basic operation* is performed by the processors: just one transposition of two adjacent elements of vector *PERM* is sufficient to generate a new permutation. Moreover, as permutations are generated only once and they are not stored in the memory to be used later, there is *no extra work* done by the network.

The network needs no Master (or Host) to indicate what processors should do step by step. Each processor only knows its position inside the network and *controls itself*. In this way we can say that our network is *modular*: for generating the permutations in ${}^{n+1}P_{n+1}$, it suffices to append a processor numbered $n+1$ as the right neighbour of processor n.

Because of its regularity and simplicity the network can be implemented in VLSI. If so we should develop means to ensure its tolerance to failures. For that we have to introduce a slight modification on the Algorithm 1 from section 2, leading to Algorithm 2 below. Now each processor sends its relative position along with the permutations. After reading a permutation p_i into vector *PERM* (with a fixed size $n+1$), a processor discovers its relative position by accessing *PERM*[0].

Algorithm 2:

```
Receive(left,PERM[0::n])
{read a permutation from the left neighbour}
WHILE PERM[1] < > 0
{there is something to read}
  SEQ
    k: = PERM[0]+1
    {get the relative position into the network}
    PERM[k+1]: = k
    PERM[0]: = k
    {save the relative position to be sent}
    SEQ i = [0 FOR k]
      SEQ
      PERM[k-i+1] <=> PERM[k-i]
        {exchange two adjacent el's}
        PAR
          Send(bottom,PERM[1::k])   output
          PERM}
        IF
          (position < > n)
            Send(right,PERM[0::n])
  {send PERM to the right neighbour}
  Receive(left,PERM[0::n])
IF
  (position < > n)
    Send(right,PERM[0::n])
    {send PERM[1] = 0 to the right neighbour}
```

Our fault tolerance hypothesis is that processors can always assure communication with their neighbours, even when their computations fail.

In a VLSI implementation PE_1 interacts with the circuit. It must receive some kind of signal from the circuit to begin computation: when the network has to start computation there is $PERM = [0, -1]$ and $PERM = [0, 0]$ read by PE_1 in `Receive(left, PERM[0::n])`.

Proposition 3:

Let fp be the number of fair processors in the network, everyone performing Algorithm 2. Then the network will generate ${}^kP_k$ $(1 \leqslant k \leqslant fp)$.

Proof

If $fp = 1$, then, as communications are assured, the vectors *PERM* received by PE_1 will be read by the only fair processor, say PE_t. So it will generate $p_1 = 1$ and stop after reading $PERM[1] = 0$. Assuming that the proposition holds for $(fp-1)$ fair processors, let us prove it for fp fair processors.

The network formed by the $(fp-1)$ first fair processors generates ${}^kP_k$, $(1 \leqslant k \leqslant fp-1)$, by the induction hypothesis.

Now, let PE_r be the rightmost fair processor in this network. The vector *PERM* sent by PE_{fp-1} will be read by PE_r, by the fault tolerant hypothesis. PE_r will then know that for each permutation read it has to intercalate $k = fp$ in it, k times, generating ${}^{fp}P_{fp}$. ■

In other words, in our network implemented in VLSI with n processors and Algorithm 2, the number of fair processors is the actual size of the network implemented with Algorithm 1.

In this way we can append fair processors to the right end of the network in order to augment its length in case of failure.

4. Ranking and unranking

Let anti-position be the position of a given element inside a permutation, from right to left. In this way the anti-position of the element 3 in the permutation 1432 is 2 and the anti-position of the element 1 in the permutation 1 is 1.

Let p be a given permutation of ${}^mP_m$ for some $m \in [1, N]$. We call p_k the subpermutation of p *in* ${}^kP_k$ for $k \in [1, m]$ and i_k $(i_k \in [1, k])$ the anti-position of k in p_k. Clearly $p = p_m$.

Now we can construct a recursive algorithm in order to compute $R(p_m)$, the position index (ranking) of a permutation of ${}^mP_m$ generated by the mth processor.

With the anti-position i_k of k we know that p_k is the i_kth permutation generated from a p_{k-1} read. We also know that we have generated $(k \cdot A)$ permutations before p_k, where A is the number of permutations generated by the $(k-1)$th processor before generating p_{k-1}, i.e., $[R(p_{k-1})-1]$.

So $$R(p_k) = i_k + k \cdot [R(p_{k-1}) - 1] \qquad (1)$$

Thus:

$$\begin{aligned} R(p_m) = i_m &+ m \cdot [(i_{m-1} - 1) \\ &+ (m-1) \cdot [(i_{m-2} - 1) \\ &+ (m-2) \cdot [(i_{m-3} - 1) \\ &+ 3 \cdot [(i_2 - 1) + 2 \cdot (1 - 1)]] \ldots]] \end{aligned}$$

E.g., what is the position index $R(p)$ were $p = 1342$?

We have $i_4 = 2$; $i_3 = 2$; $i_2 = 1$

And

$$R(p) = 2 + 4[(2-1) + 3(1-1)] = 2 + 4[1] = 6.$$

Let now r_m be a given ranking. For unranking, we want to find p_m such that $R(p_m) = r_m$. Hence we must find the anti-position i_k of every element k in p_m $(1 \leqslant k \leqslant m)$. We do that recursively again by computing i_k from r_k.

There is only one way to write r_k as

$$r_k = k \cdot r_{k-1} + q_k, \ (q_k \in [1, k])$$

and by (1)

$$R(p_k) = i_k + k \cdot [R(p_{k-1}) - 1], (i_k \in [1, k])$$

then

$$i_k = q_k \quad \text{and} \quad R(p_{k-1}) = r_{k-1} + 1$$

As an example we will obtain $p_4 \in {}^4P_4$ from $R(p) = 13$:

We have

$R(p_4) = 13 = 4 \cdot r_3 + 1 \Rightarrow R(p_3) = 4$ and $i_4 = 1$

$R(p_3) = 4 = 3 \cdot r_2 + 1 \Rightarrow R(p_2) = 2$ and $i_3 = 1$

$R(p_2) = 2 = 2 \cdot r_1 + 2 \Rightarrow R(p_1) = 1$ and $i_2 = 2$,

Then $i_2 = 2 \Rightarrow p_2 = 21$

$i_3 = 1 \Rightarrow p_3 = 213$

$i_4 = 1 \Rightarrow p_4 = 2134$

Both Ranking and Unranking algorithms can be implemented in $O(n^2)$ time.

5. Conclusion

In this paper we have presented a linear processor network for generating all the $\Sigma k!$ permutations of ${}^kP_k$ for $(1 \leqslant k \leqslant n)$. The time complexity of the proposed network is $(n+n!)$ units of time, where a unit of time is the time spent by a processor to transpose two adjacent elements and perform one I/O operation.

We programmed the network in the OCCAM parallel language and we simulated it on the FPS T-20 4-Cube parallel computer.[5] For $n = 10$ the network takes approximately 101756 milliseconds to generate all the 4.037.913 permutations it is supposed to.

As seen at section 3 the major goals stated in the Introduction have been achieved. Remarkably the little space needed and the fact that the network is regular, VLSI suitable and fault tolerant. Moreover this network can be seen as a way for traversing combinatoric trees. We can think of a combination as a succession of ones and zeroes, indicating if an element is to be selected or not in that combination. With a simple modification in the program for permutations we can generate all the combinations of k objects out of n, for $k = 1, 2, \ldots, n$. This network can then be used for solving the knapsack problem and other related problems.[2]

As there is a combinatoric explosion in the computation, a good question is how to generate the maximum number of permutations in a given time and/or with a limited number of processors. It is easy to see that the first processors have a low efficiency, so we could develop means in order to increase their efficiency.

One way is to use the processors that become idle as the time goes by. As an example, processor 1 becomes idle immediately after sending ${}^1P_1 = 1$ to its right neighbour, so it could share the work done by the busiest processor, i.e., the last one. However this strategy will destroy the network simplicity.

Another way is to take advantage of the structure of the transputer processor and consider the physical processors of the network as logical processes. If we implement one process per processor we get our network. But if we assign several processes per physical processor we can deal with a limited number

of processors and also improve the processors efficiency.

M. COSNARD, A. G. FERREIRA
LIP-IMAG,
Ecole Normale Supérieure de Lyon,
46, allée d'Italie,
69364 Lyon Cedex 07,
France

This work was supported by the 'Programme de Recherches Coordonnées C³' of the CNRS.

The second author was on leave from the University of Sao Paulo, Brazil, and partially supported by a CAPES/COFECUB fellowship, grant No. 503/86-9.

6. References

1. G. H. Chen and M. S. Chern, Parallel generation of permutations and combinations, *BIT*, **26**, 277–283 (1986).
2. M. Cosnard and A. G. Ferreira, A linear processor network for the knapsack problem, *Vector Applications for Parallel Processing Conference III*, Liverpool (GB), 23–25/08/87, preprint.
3. M. Cosnard and A. G. Ferreira, Generating permutations on a VLSI suitable network, *IMAG Research Report* RR700–I, Jan. (1988).
4. P. Gupta and G. P. Bhattacharjee, Parallel generation of permutations, *The Computer Journal* **26** (2), 97–105 (1983).
5. J. L. Gustafson, S. Hawkinson and K. Scott, The architecture of a homogeneous vector supercomputer, *Journal of Parallel and Distributed Computing* **3**, 297–304 (1986).
6. K. Hwang, F. Briggs, *Parallel processing and computer architecture*, McGraw Hill (1984).
7. Y. Kermarrec and R. Rannou, Presentation du langage OCCAM, *Bigre+ Globule* 52, 25–66 (Dec. 1986).
8. M. Mor and A. S. Fraenkel, Permutation generation on vector processors, *The Computer Journal*, **25** (4), 423–428 (1982).
9. I. Semba, An efficient algorithm for generating all k-subsets ($1 \leqslant k \leqslant m \leqslant n$) of the set $\{1, 2, \ldots, n\}$ in lexicographic order, *Journal of Algorithms*, **5** (2), 281–283 (1984).
10. S. Zaks, A new algorithm for generation of permutations, *BIT*, **24** (2), 196–204 (1984).

Announcements

28–30 MARCH 1990

1990 ACM Symposium on Personal and Small Computers, Stouffer Concourse Hotel, Crystal City, Arlington, VA.

Theme: Artificial Intelligence and Standards

The ACM Symposia on Small Systems provides an ongoing overview of the current state of the art in microcomputer technology and applications. This year's themes will deal with the convergence of microcomputer and artificial intelligence technologies and with the role of standards in the small computer field. Topics will include standards, advances in architectures and systems software, distributed systems, networks, parallelism and concurrency, office automation, human factors issues, CAD/CAM, database, software engineering, artificial intelligence and expert systems, logic and symbolic programming, etc., as they relate to microcomputer technology and small systems.

For further information contact:

Conference Chair, Prof. Elizabeth Unger, Department of Computer and Information Science, Kansas State University, Manhattan, KS 66506

or

Program Chair, Professor Hal Berghel, Department of Computer Science, University of Arkansas, Fayetteville, AR 72701

17–21 APRIL 1990

VDM '90. VDM and Z! 3rd International VDM Europe Symposium on mathematically based methods for systematic development of large-scale software. Kiel, Federal Republic of Germany. Sponsor: VDM Europe.

Contact:
Professor Hans Langmaack, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, D-2300 Kiel, F.R.G. Tel.: 49-431-8804470/71; fax 49-431-8802072.

6–8 JUNE 1990

Eurographics Workshop on Object-Oriented Graphics, Königswinter, Federal Republic of Germany

First call for contributions

Aims and scope

Object-oriented methods are proving to be particularly applicable to computer graphics – in formulating new graphics standards and in dynamic graphics and human–computer interaction. Specific computer graphics problems have also resulted in a critique of the object-oriented paradigm.

Contributions are sought which combine the latest results in object-oriented methods with original contributions to computer graphics. Participants will present their work and discuss problems and extensions with the group. Areas of interest include the following.

- Standardization and object-oriented languages for graphics
- Complex objects (e.g. objects with a hierarchy of parts)
- Dynamic type creation (e.g. new types created on the fly in CAD)
- Automatic classification of dynamically created objects
- Issues in 3-D animation and robotics (constraints, collision detection)
- AI and knowledge representation in object-oriented graphical systems
- Direct manipulation and object-oriented user interfaces

Full papers

The workshop will be limited to about 40 participants to encourage discussion. Selection will take place on the basis of full papers (up to 25 pages) reviewed by the programme committee. Please submit 4 copies to the address below. Papers will appear in the workshop proceedings. Invitations to submit revised versions for a book (in the *Eurographic Seminars Series*) will depend on the quality of the contributions.

Taking part without submitting a paper may be possible in a few cases if you submit a position paper with your view on current issues in object-oriented computer graphics. Invitations will be sent about one month before the workshop.

Schedule

31 January 1990	Deadline for full paper
12 April 1990	Notification of acceptance of paper
30 April 1990	Latest date for position papers
6–8 June 1990	Workshop
13 August 1990	Deadline for final paper (15–25 pp. camera-ready)

Venue and fee

The workshop will be held at Königswinter near Bonn, on the Rhine. The fee will be around DM 700, including accommodation, meals, evening boat excursion and reception at Birlinghoven Castle. Student participants without access to other funds could be subsidised.

Organisation

The workshop is organised by the German National Research Centre for Computer Science (GMD) and the Dutch Centre for Mathematics and Computer Science (CWI) and promoted by Eurographics, in cooperation with the German Society for Informatics (GI4.1.1 Special Interest Group for Graphics Systems, GI4.1.5 German Chapter of Eurographics).

Co-chairmen

Peter Wißkirchen (GMD) and Edwin Blake (CWI).

Information

Manuscripts and requests for information should be sent to the workshop secretary: Ms. Marja Hegt, O-O Graphics Workshop, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Tel: +31 20 592 4058. Fax: +31 20 592 4199. Email: marja@cwi.nl (uucp).