Adaptive Deadlock-free Packet Routeing in Transputer-based Multiprocessor Interconnection Networks

N. T. SON AND Y. PAKER
Centre for Parallel Computing, Queen Mary and Westfield College, University of London, Mile End Road, London E1 4NS

This paper presents a deadlock-free routeing algorithm for multiprocessor interconnection networks based on store-and-forward (S/F) communication. The adaptive nature of the method proposed encourages using light traffic paths. Furthermore it has the properties of avoiding blocked communication (deadlock), reducing communication delay time between source and destination, using efficiently message buffers as network resources and being able to control communication traffic flow from each processor of the network. The routeing algorithm has been implemented on a 64-node transputer network (T-Rack) configured as a number of well known topologies to evaluate the behaviour of the algorithm and some performance figures have been derived.

Received January 1990, revised April 1991

1. INTRODUCTION

For interconnected multiprocessor systems based on transputers or other processors such as Intel hypercube, the limitation of links per processor implies that the network topology, except for very small sizes, is not fully connected. Consequently, messages exchanged between two processors, if they are not neighbours, have to be sent through one or more intermediate processor(s). This requires a routing scheme so that at each processor incoming messages are guided by a router to follow a path towards the destination. A well-known technique for network routeing is based on the store-and-forward method; at each intermediate processor when a packet arrives it is first stored fully in the memory and then forwarded to the next neighbouring processor. The fact that this method involves only two processors at any given time in transmitting a packet makes it relatively easy to implement. However, there are stringent requirements on the minimum buffer space in each processor to be able to accommodate a number of full length packets.[1]

Many different routeing algorithms have been developed for S/F interconnection networks.[2–5] Most of these (such as refs 2 and 4) have introduced the concept of buffer classes or virtual channels, mainly to solve the problem of deadlock. The technique used provides directed paths of buffers from any source to any destination so that none of these paths contains a cycle, a condition which can cause deadlock. However, as the network size grows, this technique requires an increase in the buffer space at each processor, for instance in ref. 4 it is shown that the growth is linear with the network diameter. On the other hand, restrictions of buffer and routeing assignments in these algorithms often lead to a situation where some processor buffers are full while others are almost empty. This unbalanced usage leads to inefficient use of system resources (i.e. memory). Furthermore, the upper bound of necessary buffer size in these algorithms cannot be always satisfied since this depends on traffic generated by an application which is usually not predictable.

A more sophisticated method called wormhole[6] implemented by special hardware avoids unnecessary overheads of intermediate buffering of packets. Instead of storing a packet completely, wormhole operates by advancing the header of the packet directly from incoming links to outgoing links. As soon as a processor examines the header of a packet, it selects the next link to forward the packet. As the packet header gets shifted down the link, the rest of the packet follows and spreads out across the links between the source and the destination. It is possible that the header of a packet arrives at the destination before the last part of the packet has left the source. But if the header is blocked for some reason, the transmission of the packet is stopped along its transmission path, waiting in the network. This blocks the progress of any other packets requiring the links tied up by the blocked packet.

The cut-through method in ref. 7 is similar to the wormhole technique. It differs in that the packet is buffered when it is blocked, thus freeing the links on its path. Therefore resources (links) can be used more effectively, increasing the network throughput. However at each intermediate processor, a buffer has to be allocated for storing blocked packets. If the number of blocked packets is high, then allocation of buffers could require huge amount of memory. The overheads for allocating and reallocating buffers is relatively high. This can become a crucial factor to degrade system performance.

An effective and reliable multiprocessor interconnection network requires a routeing algorithm that is able to avoid deadlock and to use efficiently the existing communication links and buffers. This paper presents an adaptive routeing algorithm which controls the transmission of messages through the network. Due to influence of messages en route on others as the congestion builds up, the algorithm tries to reduce the blocking of traffic by using adaptively the existing idle buffers in the network. The traffic is not always routed over a single path, i.e. shortest path, therefore, local conditions can sometimes influence traffic patterns over a wider neighbourhood and, thus contributing to the global traffic flow. The method described, called adaptive deadlock-free routeing (ADR), ensures deadlock avoidance and effective use of system resources.

In Section 2 we first introduce a router model. We then describe the basic structure of the routeing algorithm in Section 3. Section 4 presents the architecture of the routeing module, which implements the method intro-

duced in Section 3. The problem of deadlock prevention is discussed and a constructive proof has shown the correctness of the algorithm in Section 5. In Section 6 a discussion is presented on the routeing algorithm as implemented and tested on a specific transputer machine, the T-Rack.[10]

2. OVERVIEW OF THE ROUTER MODEL

The router model is shown in Fig. 1. It contains three functional units: (1) the input unit, (2) the routing unit and (3) the output unit. All the functional units handle transmission of packets.

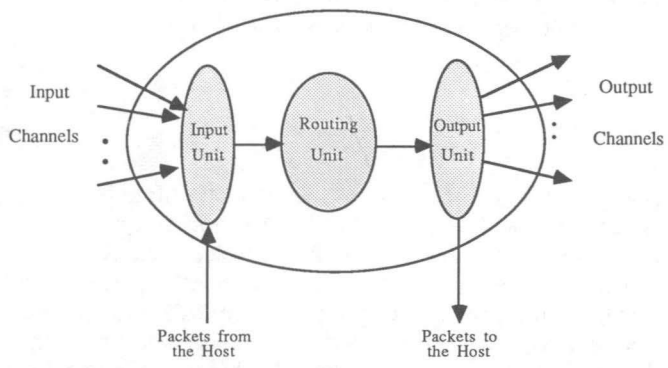


Figure 1. Router model of a S/F node.

Packets from input channels are stored in the input unit. Only one buffer is reserved for packets coming from all the input channels. The input unit distinguishes two kinds of packets: (1) transit packets which come from neighbouring processors and (2) entry packets which come from the local processor (host).

The packet buffered in the input unit is immediately routed. After processing in the routeing unit, the packet is passed to the corresponding output channel leading to an adjacent processor or to the host. This is done by consulting the routeing table stored in the routeing unit as a vector which maps the current and destination processor addresses to the output channel, i.e. $R: N \times N \rightarrow C$, where N is the number of processors and C the set of output channels.

The output unit contains a set of output channels connected to the neighbouring processors and one to the local processor. Each output channel is modelled as a first-in-first-out (FIFO) queue.

3. THE ADR ALGORITHM

A routeing algorithm provides rules that are used to transmit messages or packets through the network from a source to a destination processor. Based on how routeing varies with traffic conditions, a routeing algorithm is classified as a deterministic or an adaptive one. With deterministic routeing strategies, the path for any source–destination pair is determined *a priori* which is independent of existing traffic. Messages are forced to move in a single determined direction (such as the shortest path as used in this paper). Adaptive routeing strategies provide an alternative whereby messages could follow different paths, depending on the prevailing circumstances such as the level of network traffic, failed links or extreme traffic congestion. Basically, the ADR algorithm is based on a combination of deterministic and adaptive routeing. The algorithm is deterministic for light-traffic and becomes adaptive for heavy-traffic conditions.

Before we describe the ADR algorithm the following properties are assumed:

Assumption 1. The network contains a finite number of processors and it is connected (each processor is reachable from any other processor). Links connected to adjacent processors are bi-directional.

Assumption 2. All processors, communication links, etc. are functioning properly so that no message can be lost.

Assumption 3. Messages are transmitted as complete units: each message fits into one unit of buffer. Packetising and re-assembly are functions of higher layers of network protocols, which fall outside the scope of this paper. Unless explicitly stated otherwise, 'message' and 'packet' are used interchangeably.

Assumption 4. A message that arrives at its destination might not be consumed immediately due to two reasons: (1) the consume rate is smaller than the arrival rate and (2) there is no synchronisation between the producer and the consumer so that packets can arrive at their destination before they are needed. However, all such packets must be consumed eventually.

The simplified listing of the ADR algorithm is given below, where $d \cdot a$ represents the destination address of a packet, the link l is obtained from the routeing function $R\{n\}$ where n is the destination, and $q(i)$ stands for the state of queue i, i.e. full or not full. The adaptive routeing algorithm in OCCAM style is presented below.

```
WHILE running
  SEQ
    ...Input a packet
    ...Find link l:=R{d·a}
    IF
      q(l) NOT full
        ...Send the packet to queue l
      TRUE
        SEQ
          ...Find the q(i) that is NOT full
          ...Send the packet to queue i
```

The ADR algorithm must satisfy the rules as listed below:

Rule 1. A new packet from a local host can enter the network (local input unit) if and only if there is at least one output channel queue at its local output unit which is not full. This means

$$\sum_{i=1}^{n} q_i < \sum_{i=1}^{n} Q_i \quad \text{or} \quad \sum_{i=1}^{n} q_i \leqslant (B-2),$$

where q_i and Q_i are the current and maximum queue lengths respectively and B is the total buffer capacity in each processor ($B = \Sigma Q_i + 1$).

Rule 2. If packets arrive from a number of input channels at the same time, then they are handled by a scheme to ensure fairness.

Rule 3. A transit packet has higher priority than an entry packet.

Rule 4. If the local host is not ready to receive a packet destinated to it, then it is sent to the output channel queue with the smallest queue length.

Rule 5. If the queue of an output channel determined by the routeing function $R\{n\}$ is full, then the packet is sent to the queue of the output channel with the smallest current queue length.

Rule 6. If there is more than one queue with the smallest current queue length, one is chosen randomly.

From the above rules, it follows that messages try first to follow the path as defined by the routeing function, such as the shortest path. According to ref. 9, for networks which consist of homogeneous processors with links of equal bandwidth, for light message loads, the choice of shortest path between any source and destination gives the optimal performance. The adaptive strategy is used only when a shortest path starts getting overloaded or a destination processor is temporarily unable to consume incoming messages.

4. ARCHITECTURE OF THE COMMUNICATION ROUTER

The architecture of the communication router is defined by its logical components, their characteristics and interfaces. As shown in Fig. 2, a communication router consists of the following modules.

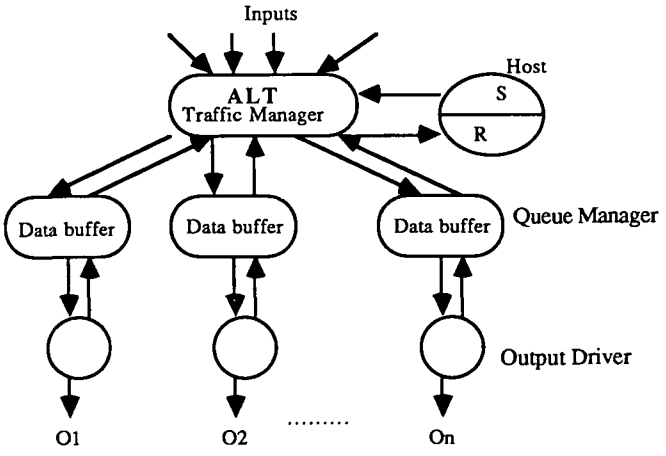


Figure 2. Communication architecture of a processor.

1. *Traffic Manager* implements the ADR communication algorithm. The Manager controls the current state of the router and synchronises the activities of other modules in the router.

2. *Data Buffer* is organised as a queue structure. One queue is used for each link. Packets are stored in queues before they are transmitted to the next processor. Handling of packets in each queue is controlled by a queue manager.

3. *Output Driver* removes packets from queues and transmits them via the links. It provides the same effect as an output guard.

All modules run in parallel and they refer to one another by means of internal communications (channels).

4.1. Traffic manager

The traffic manager is the main part of the router whose task is to receive packets arriving at input links or from a user process (host) and to channel them following the routeing strategy. The flowchart given in Fig. 3 presents the semantic function of the traffic manager.

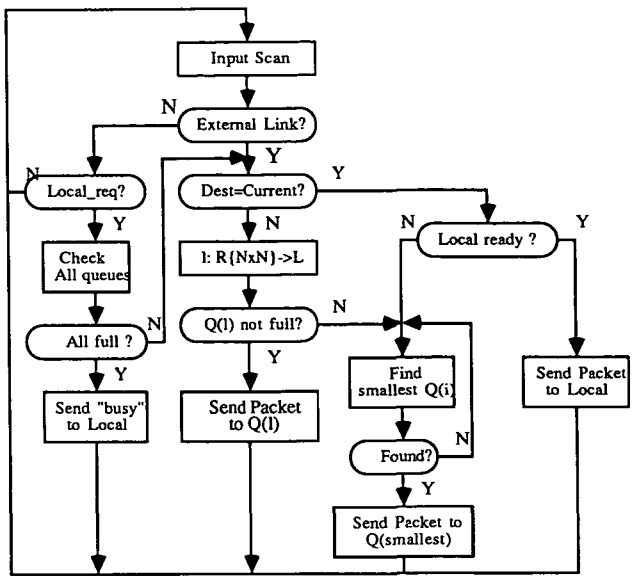


Figure 3. The traffic manager.

At a given time only one packet can be handled by the traffic manager. Unlike many other S/F communication networks, there is only one buffer for all input channels and the host input. This input buffer handles one packet at a time. Therefore a packet is accepted only if the previous one has been removed.

For an incoming packet when the traffic manager finds the routeing information from the routeing function R: $\{n\} \rightarrow$ link, it sends a request to the corresponding queue manager to ask for a place in the queue. The response is a value of the current queue length. If the queue length is smaller than the maximum queue size allowed for each link, then the packet is sent to that queue and another packet can be received by the traffic manager. A fairness structure using a round-robin technique[8] has been implemented to avoid the starvation in the case whenever there is more than one packet waiting in the link inputs at the same time. When the queue length of the desired queue reaches its maximum, then the adaptive nature of the algorithm becomes effective. In this case the traffic manager sends requests simultaneously to other queue managers. A queue with the shortest queue length, if it exists, is chosen. A critical situation arises when all queues reach their limits. In this case, the packet is kept in the traffic manager until there is space in one of the queues.

4.2. Queue manager

The queue manager (Fig. 4*a*), is the module used to control the buffering of packets. The number of current packets buffered in a queue is updated when a packet is received from the traffic manager or a packet is taken by the output driver. A circularly linked list has been used for the data structure.

4.3. Output driver

This module is used as an interface to the external link. It requests the queue manager for a packet each time (Fig. 4*b*). If there are any, then it takes one and tries to send the packet down the link. As soon as the link is ready the packet is sent and the output driver requests

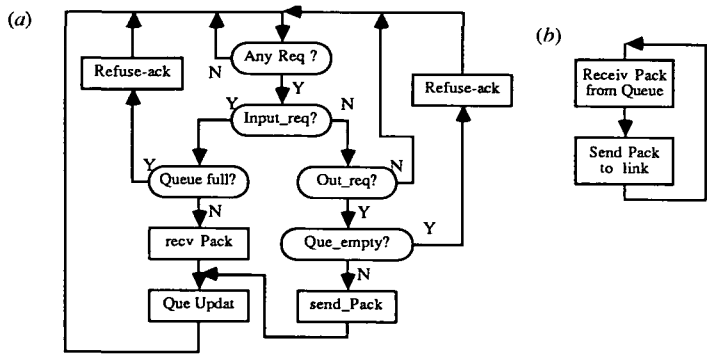


Figure 4. (*a*) The queue manager and (*b*) the output driver.

the queue for another one. If the link is not ready the packet is kept in the output driver until the link becomes ready. During this waiting time no packet can be taken out of this queue. The queue may be filled by new incoming packets until the traffic manager realises that this queue is full and no more packets are accepted.

5. VALIDATION

5.1. Correctness issues

For communication networks deadlock implies a situation in which: (1) no packet can be forwarded due to all buffers being full of packets waiting in a cyclic fashion, or (2) there are still packets moving in the network, but can never arrive at their destinations. This is referred to as livelock.

Fig. 5*a* shows a very simple example of a S/F deadlock case (called direct S/F deadlock[4]) in which given two adjacent processors A and B, A's buffers are full with packets destinated for B, and B's buffers are full with packets destinated for A. As a result, no packet can move.

Another more complicated deadlock example is shown in Fig. 5*b*. In this case, more than two processors are involved in the deadlock. The packets are forced to move in the direction of a cycle. Each processor obtains a full buffer of packets, which are destinated to the next processor in line but one. No packet can advance toward its destination, thus causing deadlock.

The S/F deadlocks given refer to the situation where there is a cycle of the buffer requests among a set of communicating processors, all of them having no empty buffers left. One of the main reasons is the input load probably exceeds the network capacity and the routeing algorithm forces a single direction on a packet, causing blocking of communication links. Most routeing algorithms assume that a packet which arrives at its destination will be consumed immediately. But, in practice this is not always true. Consumption of packets depends so much on user applications which run asynchronously in each processor. To store all the incoming packets while consumption rate is falling behind requires increasingly large memory for each processor which can easily exceed the available size which is only few Mbytes for transputers. If message communication is not well synchronised, this becomes one of the reasons causing communication blocking.

From Section 3 describing ADR algorithm, three important points have to be stated: (1) the algorithm

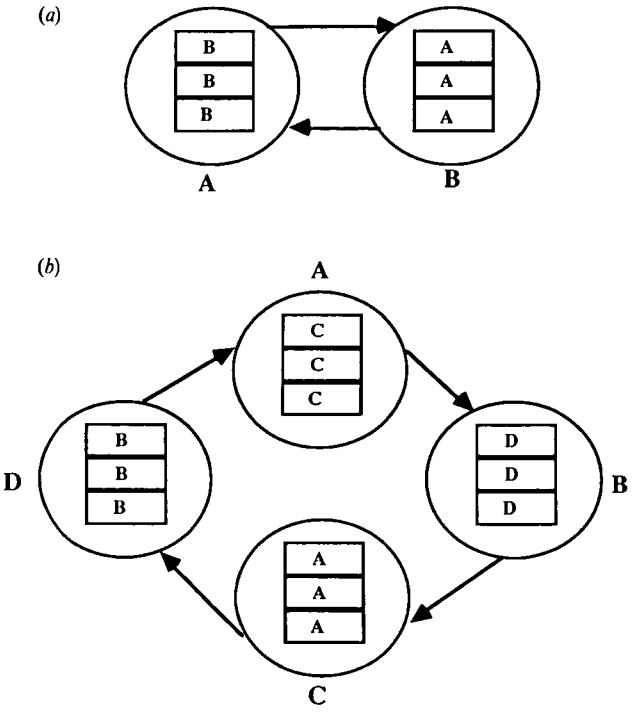


Figure 5. Deadlock situation. (*a*) Direct deadlock; (*b*) undirect deadlock.

simply prohibits the generation (but not passing) of a packet into the last empty buffer at a processor; (2) packets are not restricted to move along any fixed path; wherever they are blocked, they try to find free space in the neighbourhood in order to get to their destinations; and (3) packets coming from different inputs to a processor are always accepted in a certain order. To clarify the algorithm, the examples given below are chosen which are usually deadlock prone.

Example 1. Two-processor communication: one sends and the other one receives.

As shown in Fig. 6*a* the user process in A sends packets to B and B receives packets from A. Supposing that due to some reason, for instance consumption of packets in B is slower than sending of packets in A, these are not accepted by B and therefore they are buffered in the output channel queue in B to be sent back to A. If user in A keeps sending packets, then after a certain time all buffers in both processors will become full. Notice, however, that the router prohibits the user in A introducing a packet into the last empty buffer, hence at least one empty buffer must exist in A. We call this case where only one empty buffer exists the critical situation. Those packets which are sent back to A are treated in A as transit packets. These packets still keep moving between A and B. Packets returning to the sender act as a brake on the incoming traffic generated by the sender.

Example 2. Two-processor communication: each sends and receives concurrently.

In this case the two processors (A and B) execute concurrently two tasks, namely send and receive packets. Packets to be transmitted at each processor are placed into the queue of the output channel leading to the other processor. For any reason as in Example 1, to avoid blocking, the packets refused are sent back and since the

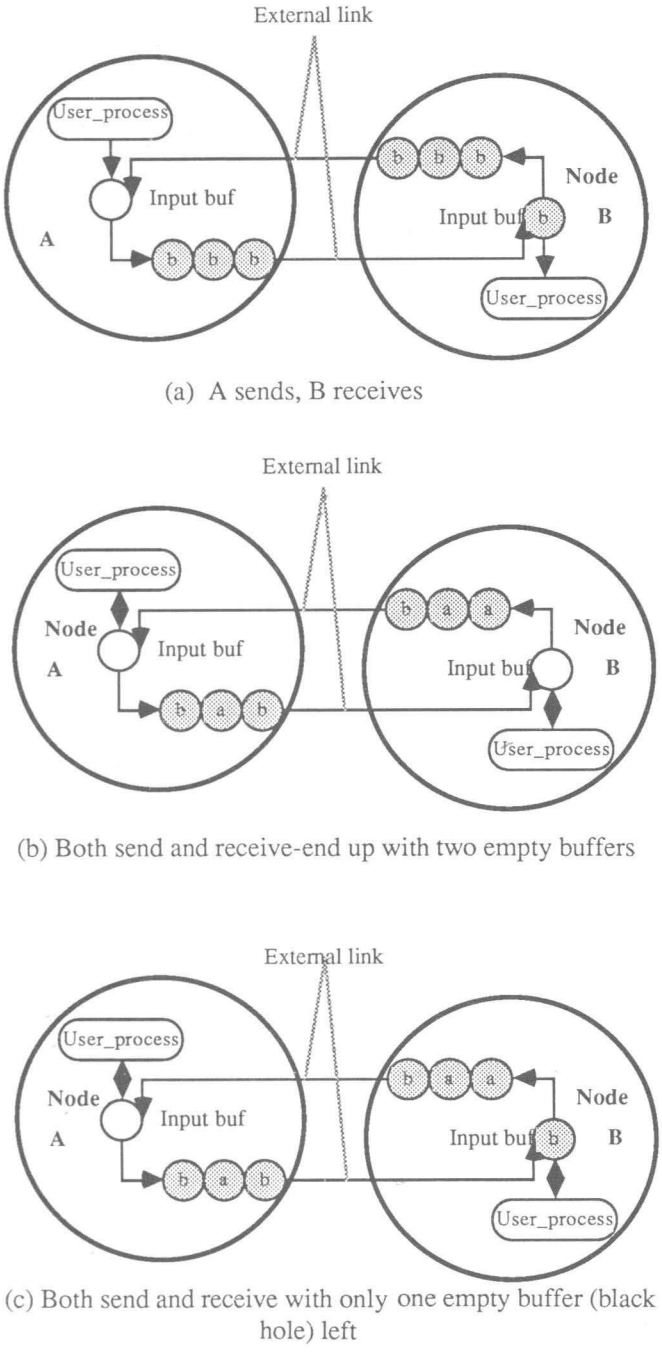


Figure 6. Critical situations between two adjacent nodes.

source is unaware of this it keeps sending packets until all queues are full. The result is each processor ends up with one last empty buffer as shown in Fig. 6*b*. This does not remain so for long, since priority is given to the transit packet, there is a moment when, say, in processor B a packet is received from the input channel, freeing one buffer in the output channel queue of the processor A and therefore, one more packet can enter the network resulting only one free buffer as shown in Fig. 6*c* causing the critical situation. We call the last empty buffer black hole. Suppose that there are only two processors in the network, then the black hole is forced to hop along a cycle, forcing packets to move along the same cycle, in the opposite direction.

Example 3. An irregular network.

In case of more than two processors in the network one can refer to the example of an irregular network as shown in Fig. 7*a* this is a more complex case where more than two processors are involved. These processors are connected in an irregular topology which represents a general case provided that the Assumption 1 (connected) is valid. For simplicity we assume a maximum queue length of one for each output queue. The communication requirement is random and the rate of packet entry is higher than the consumption.

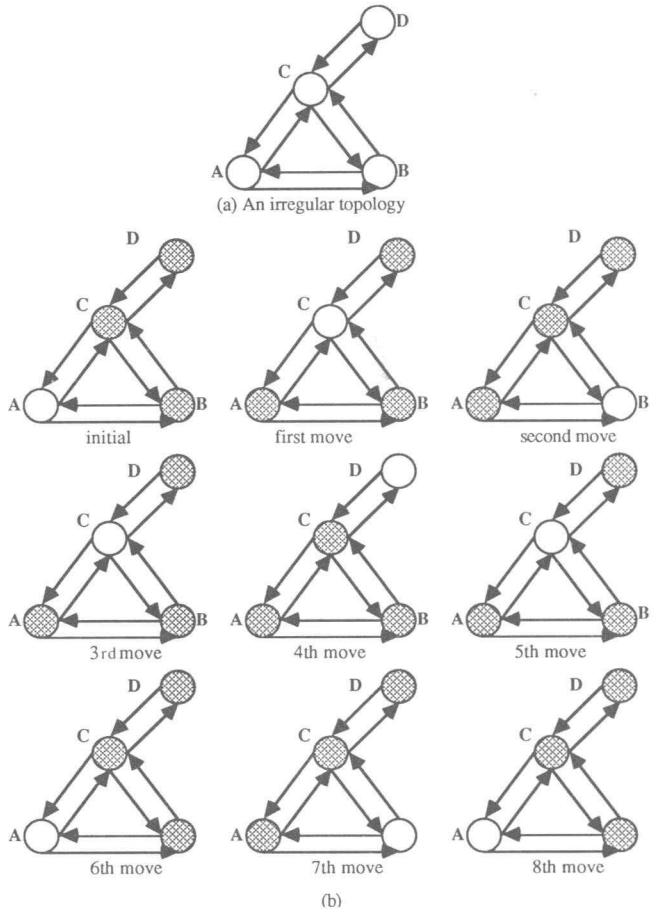


Figure 7. The trajectory of the 'black hole' in an irregular topology.

One can trace the network and finally find out that after a certain time, the critical situation arises with the black hole. The fairness structure provided by the algorithm determines the path along which the black hole moves. Connectivity and fairness are sufficient conditions to show that the black hole will move around and visit each processor at least once. Fig. 7*b* illustrates the trajectory of the black hole for the topology given in Fig. 7*a*.

From the two examples described above, it is clear that the input unit of the router simply prohibits the acceptance (but not transit) of a packet into the last empty buffer (input buffer) at a processor, hence at least one empty buffer must exist somewhere in the network. The empty buffer hops from node to node around the network causing the packets not to follow the shortest path but the path in the opposite direction to that followed by the black hole.

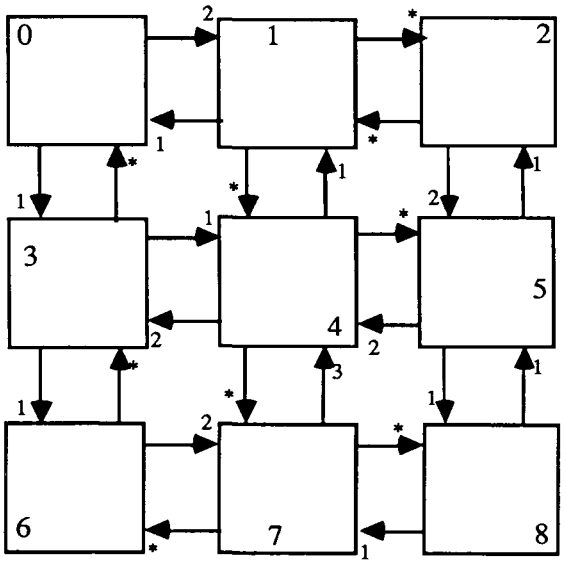
Let us say, the situation is that only one empty buffer

exists somewhere in the network and moves in the network according the fairness rule.

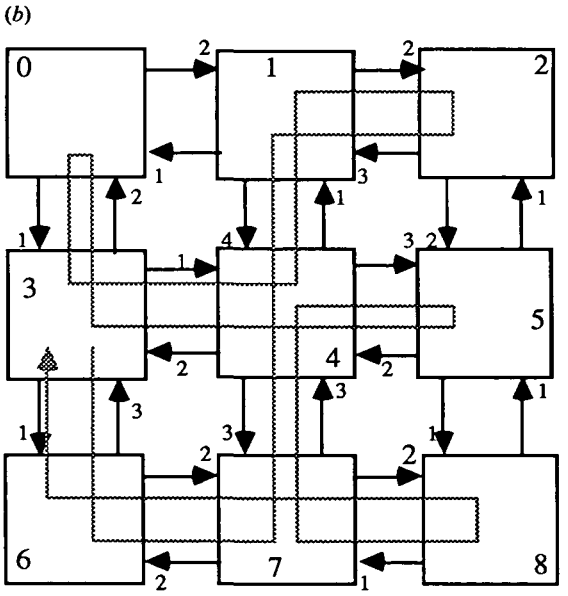
5.2. A proof for ADR algorithm

Let us consider a network as a graph $G = \{X, A\}$ where X is a set of communication processors and A is the set of communication links. G is considered a connected graph and links connecting two processors are bidirectional. In this case the move of the empty buffer is equivalent to the problem of finding one's way in a maze. This is a classical problem in graph theory where a person gets lost in a maze and does not know the maze plan. However, it is still possible to find the exit, if one follows the simple rule that one never goes twice in the same direction along any one path (link) and at the junction (node) one takes the path from which one first arrived at the same junction only when no other choice is available. This ensures that following these rules, one traverses all paths so that each junction will be visited until the exit is found as proven theoretically in ref. 16.

(*a*)



(*b*)



(*c*)

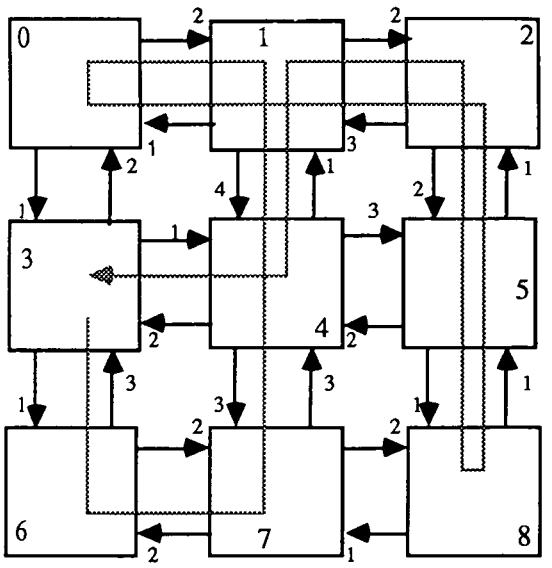


Figure 8. Illustration of black hole move. (*a*) State before the block hole occurs in node 3. (*b*) First round. (*c*) Second round.

The ADR algorithm, in particular Rule 2 ensures that the empty buffer moves along a path which covers all links at least once and comes back to the starting processor. To illustrate, Fig. 8*a* shows the state of a network at the moment just before the black hole occurs. Each input channel of a processor is assigned with an index not greater than the total number of input channels of this processor. These indices express the order in which the packets from neighbours enter this processor. The asterisk (*) indicates the channel from which the next packet will be accepted. Supposing that the black hole occurs in processor 3, then Fig. 8(*b*, *c*) shows the two rounds of the path, traversed by the black hole, covering all the links and also all the processors of the network. (Note that the black hole moves in the opposite direction of packets.)

If we put all the links in the order they are traversed, then we have a circuit of all links of the network. Imagine that packets are stored along these links. Since the empty buffer moves along this circuit in one direction, packets certainly move in opposite direction so that they also visit each node of the network until packets reach their destination.

In order to prove that deadlock will not occur, the following definitions mentioned in ref. 6 will be used:

Definition 1. A channel dependency graph $D = G(C, E)$ is obtained from a given interconnection network I with its routeing function R and the messages to be routed according to this function. D is a directed graph where each vertex c_i of D ($c_i \in C$) corresponds to one channel of I if and only if there is a message to be routed via this channel. An edge of D between c_i and c_j exists if c_i and c_j are adjacent in I and there is a message in c_i which is to be forwarded to c_j according to the routeing function R.

Definition 2. A deadlocked configuration for a routeing function R is a configuration of the channel dependency graph where there exists a cycle along which

$$\forall c_i \in C, \quad (c_j = R(c_i, n) \mid size(c_j) = cap(c_j)).$$

Above $size(c_j)$ denotes the number of messages in the queue for channel c_j and $cap(c_j)$ denotes the capacity (maximum queue length) of channel c_j.

In this configuration no message can advance because the queue for the next channel is full. A routeing function R is therefore deadlock free if no such deadlock configuration occurs which means that there must exist at least one channel with the channel capacity exceeding the current channel requirement (free buffer).

Assertion. The ADR algorithm is deadlock free.

Proof. Suppose the ADR algorithm leads to a deadlock situation. This means that packets move to wherever there are empty buffers after attempting unsuccessfully their shortest paths, since queues on shortest paths are full of waiting messages. The resulting network configuration is such that the queues of all channels reach their capacity and hence, the condition in the Definition 2 is satisfied. This however contradicts the Rule 1 (Section 3) which ensures that there is at least one empty buffer in the system. In this case the routing function is determined only by the move of that empty buffer which has been shown above to traverse along a circuit covering all the channels at least once. Therefore, in this configuration, there always exists one channel with the capacity exceeding the current channel requirement. Messages will move through all channels (all nodes) at least once and they will eventually be consumed. This proves that the ADR algorithm is deadlock free.

The ADR algorithm so far has been demonstrated to be deadlock free. However it is not free of livelock which is a situation where one or more packets are never able to reach their destinations. As long as packets are able to move along their shortest paths between the source and destination, it is obvious that they will definitely reach their destinations. Due to the adaptive nature of the algorithm, it may happen that at each node packets (i.e. livelock packets) arrive, links leading to the packet destination are saturated so that packets must follow the other way which never reaches the destination. Fig. 9, below, illustrates such a case.

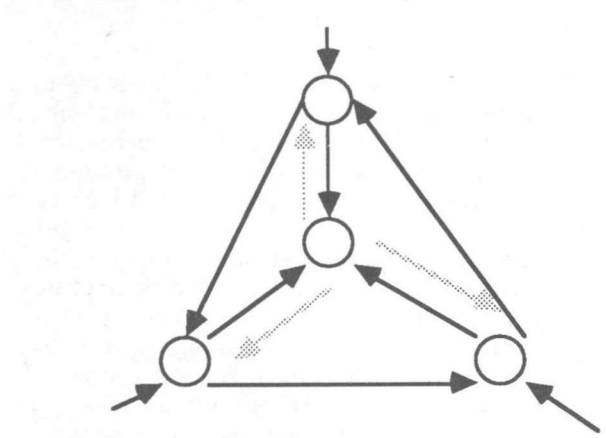


Figure 9. A livelock case.

The central node is the destination of the packet. Assume that when the packet arrives at one of the three surrounding nodes, the link leading to the central node has just been filled up by a new packet or any transit packet from somewhere else. It has to move to the next node to which the link is still free and so on it never reaches its destination. In reality, this would not happen in the network containing communication processors which run asynchronously. However, to ensure that this is avoided one can introduce a path length counter to the packet header. The path length counter is initialised as 0 by the source and incremented each time when a packet arrives at a node. Those packets whose path length counter is greater than a prescribed constant L ($L > 2*r$, where r is the number of total links in the network) are delayed in each node as they go through until the link on the path leading to their destination accepts them or a time which is proportional to the value L is reached. This, therefore, enables all packets to reach their destinations eventually, meanwhile the adaptive nature of the ADR algorithm is still preserved and livelock is prevented.

6. IMPLEMENTATION

6.1. System overview

The ADR algorithm has been implemented on a target system (T-Rack) shown in Fig. 10 which is a network of 64 transputers.[10] Each transputer has four links to connect with others. Since the links are autonomous DMA engines,[11] the processor is free to perform computation concurrently with link communication which makes it particularly suitable for implementing the S/F technique. Different desired connection patterns can be generated on the system by using the switch facility.[12] The network is connected to the host, a SUN 3/160 system, via an interface transputer board (Tadpole).

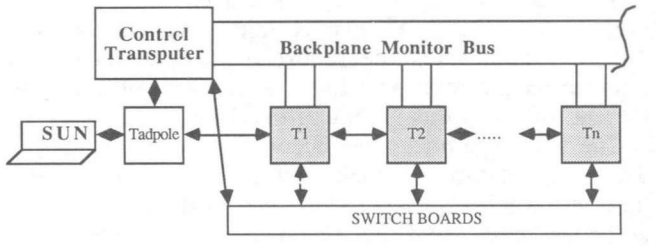


Figure 10. T-Rack architecture.

There is also a facility to access each individual transputer through the backplane bus from the control transputer. This facility is used for monitoring and synchronisation of transputers without affecting the communication performance of the system. The main task of the control transputer is to set the array of switches of the crossbar switch so that a desired topology of the network can be obtained. This is done before the routeing process is loaded together with the user process from the SUN host to the individual processor of the network. This allows setting different types of topologies for experiments shown.

To set up the system, a user generated information file, which defines the transputer network topology, is required. A facility on the SUN host allows determining information from this file, about: (1) the system configuration for setting switches and (2) the routeing table of an $N \times N$ matrix. Each row of the matrix defines the routeing function for corresponding processor to any others. The initial routeing strategy is chosen, depending on the purpose of communication. There are two alternative strategies being available in our system, i.e.

the simple Floyd's shortest path technique[13] and the balanced shortest path technique.[14] The system initiation includes setting switches for topology selected and loading the routeing tables.

The transputer architecture supports any number of concurrent processes sharing the processor time so that the routeing software can run as procedures executed concurrently with the user application in a transputer. To communicate the user needs only to call corresponding procedures (**send**(), **receive**()) serving as system communication functions. The format of a communication packet is defined as follows:

```
typedef struct    {
  unsigned char DEST      ;
                SOURCE    ;
                BLOCK_NUM ;
                DATE_TYPE ;
                LENGTH    ;
           char *Block    ;
                packet    ;
```

There is no need for user to know how packets reach their destinations.

6.2. Experimental implementation

The ADR algorithm has been implemented and tested on the T-Rack and also on the PARSYS supernode machine. Some experiments have been carried out on several topologies shown in Fig. 11. There are two additional processes called **report**() and **sample rate**() which are used to monitor the communication activities in each processor such as the arrival rate, the current length of each output queue, etc. After a certain time interval, the sample rate invokes the report process to record all monitoring information which have to be collected. At the end of a run this is sent to the SUN host.

The first experiment has been carried out to test the deadlock situation. To create heavy traffic, for each valid link the outgoing queue has been assigned one buffer unit and the input load has been made to exceed the consuming capacity by adding a delay in the receiving processes. Each processor sends packets to others in a uniform manner. The test program has been run for different topologies with increasing input loads. Communication in all cases has been successfully completed and the results have shown that no deadlock occurs even for the critical case.

For the deterministic routeing algorithm the deadlock state does not arise so long as there are enough buffers in queues. The number of necessary buffers increases rapidly with the input load. An experiment has been carried out where the output queue of each link has been assigned with the maximum length of eight. The input load of the network is defined as follows: for each unit of the input load each processor sends packets to all other processors, which means if n is number of the processors, then there are $n^*(n-1)$ packets entering the network at a time. This procedure has been repeated for a number of times. After a certain time, one observes none of the packets are able to move. Using monitoring facility allows us to find out where the deadlock has occurred.

Fig. 12 shows the measurement results of communication throughput in different topologies, namely ring, binary tree, 2D mesh, cross mesh, and hypercube, using the ADR algorithm and the same input load as mentioned above. For each topology, the curve first rises since the input load of the network is small. If the input load increases, all the links reach the maximum capacity, thus the throughput is then saturated. Among these topologies, the cross mesh is found to be the best one, better than the hypercube, since the cross mesh has a diameter less than the cube (diameter of cube is 4 and of cross mesh is 3) and there is almost no difference of the average distance between them (average distance of cube is 2.20 and of cross mesh 2.22[15]).

Comparing the binary tree with the ring, the former has a communication throughput worse than the latter, although it has smaller diameter (diameter of tree is 6 and ring is 8). The reason is due to the difference of the average distance between them (tree 5.089 and ring 4.25). For a given size topology, the less the average distance, the more alternative paths it contains. Clearly, our routeing algorithm is more suitable for topologies with several alternate paths for every pair of processors.

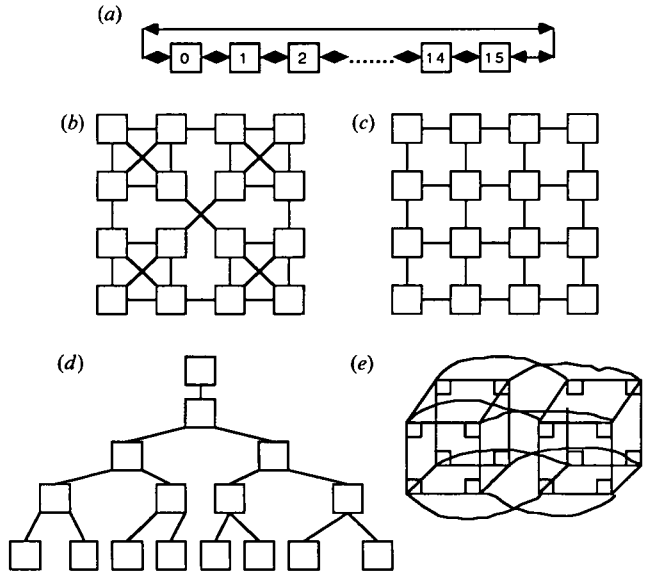


Figure 11. Some typical topologies. (*a*) Ring; (*b*) cross mesh; (*c*) 2D mesh; (*d*) binary tree; (*e*) hypercube.

Fig. 13 shows the number of packets arriving at their destinations which has been measured in unit time. The input load is lighter than in previous experiments. The result also suggests that the hypercube gets more packets to their destinations in a unit time and the tree which is the worst one.

From the above experiments one can say that in the design stage of a multiprocessor network like transputers, using the adaptive routeing algorithm, the cross mesh is the most suitable topology due to following reasons: (1) the network is extendable with fixed number of processor links; (2) it has small average distance and therefore (3) there are more paths between any pair of processors paving packets moving fast in the network.

6.3. Communication performance

An analysis about the impact of the routeing overhead is being prepared as another publication. The routeing algorithm has been tested with different network traffic loading offers and also compared with other common

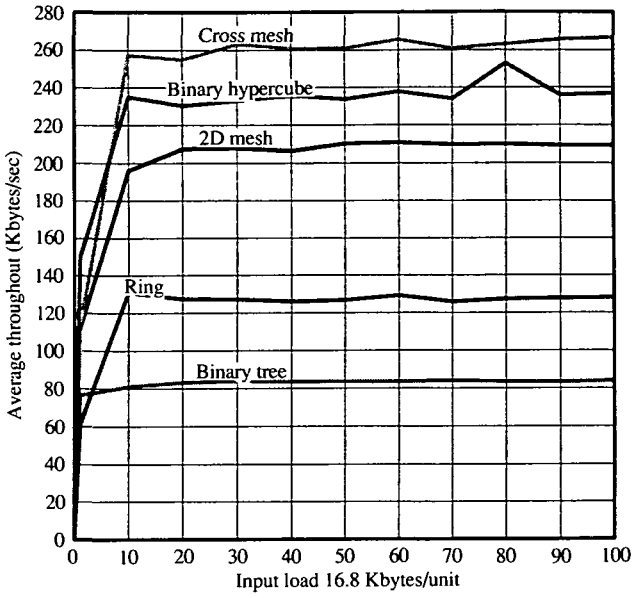


Figure 12. Network throughput versus input load for different topologies.

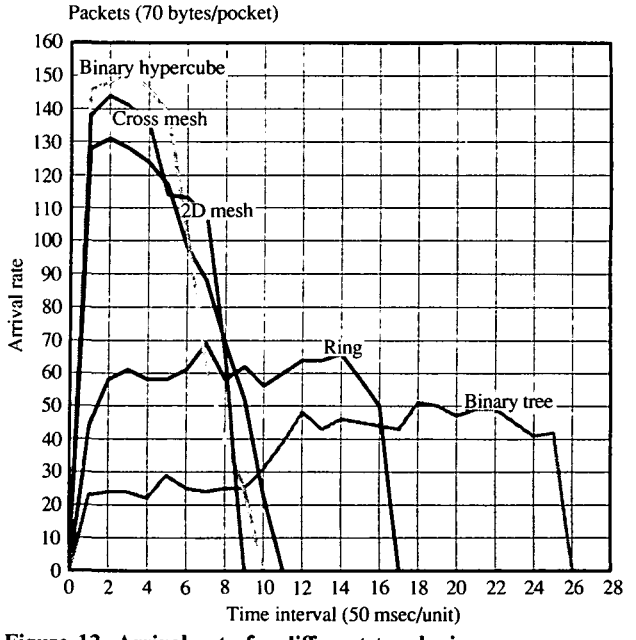


Figure 13. Arrival rate for different topologies.

routeing algorithms in terms of performance such as average packet delay, and network throughput. The results given in Fig. 14 show the average normalised network throughput versus the network traffic load of the ADR algorithm and the deterministic shortest path

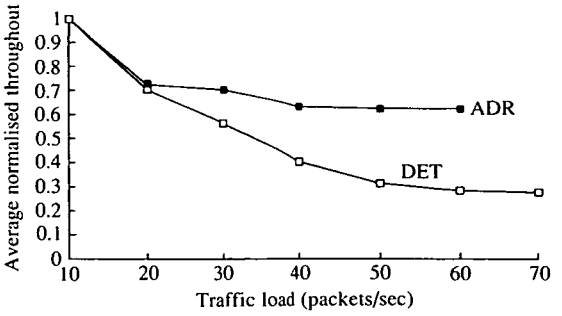


Figure 14. Throughput versus traffic load in two different routeing algorithms.

algorithm respectively. The experiment has been carried out on the PARSYS supernode machine, which contains 16 transputers (T800). The traffic load is defined as the number of packets sent from every node i to every node j per second (γ_{ij}), which is an exponential distribution function. Looking at the figure, the ADR algorithm shows its advantage against the deterministic one at the high rate of the network traffic load.

7. CONCLUSIONS

We have shown a simple deadlock-free routeing algorithm which can be constructed and applied for any arbitrarily connected communication network. The algorithm uses a dynamic routeing strategy that exploits the possible different paths that exist between a source and a destination so that the existing resources (both links and buffers) can be used effectively. The correctness of the algorithm of being free of deadlock has been constructively shown and verified by implementation. Furthermore, results obtained from measurements have shown the efficiency of this algorithm over the deterministic one, in particular when the network becomes heavily loaded. The ADR routeing algorithm program is easy to integrate to application programs written to run on different types of transputer networks.

Acknowledgements

The first author expresses gratitude to Unesco/UNDP for their fellowship support. The work was carried out within the framework of the Alvey–Parsifal project and the Esprit Parallel Application Programme. The authors wish to acknowledge the support of the Polytechnic of Central London and the Centre for Parallel Computing, Queen Mary and Westfield College, London. Many thanks to Mr Malcolm J. Shute for the informal discussion and special thanks to the referee who made a number of useful comments on this paper.

REFERENCES

1. J. Yantchev and C. R. Jesshope, Adaptive, low latency, deadlock-free packet routing for networks of processors. *IEE Proceedings*, **136**E, (3), 178–186 (1989).
2. K. D. Gunther, Prevention of deadlocks in packet-switched data transport systems. *IEEE Trans. on Comp.*, **C-29**, 512–524 (1989).
3. Alan Knowles and Todor Kantchev, Message passing in a transputer system. *Microprocessors and Microsystems*, **13** (2) (1989).
4. P. M. Merlin and P. J. Schweitzer, Deadlock avoidance in store-and-forward network. *IEEE Trans. on Comp.*, **C-28**, 345–360 (1980).
5. Cheung Wing Chan and Tak Shing P. Yum, An algorithm for detecting and resolving store-and-forward deadlocks in packet-switching networks. *IEEE Trans. on Comp.*, **C-35** (8), 801–807 (1987).
6. W. J. Dally and C. Seitz, Deadlock free message routing in multiprocessor interconnection networks. *IEEE Trans. on Comp.*, **C-36** (5), 547–553 (1987).
7. Kermani Parviz and L. Keinrock, Virtual cut-through: a

new computer communication switching technique. *Computer Network*, **3**, 267–268 (1979).
8. Occam User Group Newsletter, no. 10 (Jan 1989).
9. Vijay Ahuja, *Design and Analysis of Computer Communication Network*. McGraw-Hill Book Company (1982).
10. A. E. Knowles, *Specification for T-Rack* PSF/MU/87/AEK/3 Internal report, Department of Computer Science, University of Manchester (1986).
11. INMOS, *Transputer Technical Reference* Prentice Hall (1989).
12. N. T. Son, An Approach to the dynamic Reconfiguration of a Multi-transputer Network. PSF/PCL/WP6/89/3 Parallel Computing Research group, Polytechnic of Central London.
13. Nicos Christofides, *An Algorithmic Approach*. Academic Press, London, New York, San Francisco.
14. M. Bozygit, A dense variable topology multicomputer system. Ph.D. Thesis submitted at Polytechnic of Central London.
15. Dharma P. Agrawal *et al.* Evaluating the performance of Multicomputer Configurations. *Computer*, **19** (5), 23–37 (1986).
16. Clifford Wallace Marshall, *Applied Graph Theory*. Wiley Interscience (1971).

Correspondence

Does a Point Belong to a Polygon?

Sir,

To describe a space region when solving a boundary problem I wrote a program determining whether a point belongs to a polygon specified by the coordinates of its own vertices. My algorithm when compared with similar ones, in particular with that by R. Francline,[1] turns out to be faster. To my knowledge, such an algorithm has not yet been published. Its essence lies in the following.

In the given polygon, one vertex is singled out, and the remaining ones are connected to it. Any two adjacent vertices and the one singled out make a triangle. The point may belong to one or several triangles or to none of them. If the number of triangles to which the given point belongs is even, then the point lies outside the polygon, otherwise it lies inside it. The algorithm seems at first sight to be more cumbersome than the traditional one, based on counting the number of the points crossing the polygon boundary by a ray drawn from the point in any direction, and not well suited for writing an efficient program. However, the comparison shows that this is not the case.

If a polygon has N vertices and, consequently, N sides, then, according to the traditional algorithm, one needs N tests to find out whether the chosen ray crosses the line, a part of which belongs to the polygon boundary. On the average, the number of these crossings should be $N/2$; moreover, one should check whether the crossing point belongs to the boundary, i.e. whether it lies between the vertices. Then the average number of basic checks will be $\frac{3}{2}N$.

In the algorithm proposed, $N-1$ checks are made to find out on which side of the rays, emerging from the given vertex and passing through the other vertices of the polygon the point lies. If the point lies between two rays, one checks whether it is inside the triangle. The number n of such checks is seldom more than 3 and, on the average, is less than 1. (The figure presents the rare difficult case.) The number of basic checks is about N.

The necessity in additional checks for 'special cases', when the point is on the boundary or on the line belonging to it, or in the polygon vertex, increases the ratio of the number of checks. In practice, the program presented in the Appendix runs almost twice as fast as the one by R. Francline.[1]

I mention the testing technique since it is right at the point where we encounter discrepancies when estimating these algorithms. The running time of the complete test program consists of the running time τ of the program to be tested plus the running time t of the remainder of the test program. Therefore if the test program runs 2.1 times faster with the program proposed here included, as compared with the case when Francline's program is included, this does not necessarily mean that their efficiencies differ by more than two times. Here we have

$$(t+\tau_F)/(t+\tau_K) \sim 2.1.$$

If in the same test program we have two calls to the program to be tested rather than one, it is quite possible that one can obtain

$$(t+2\tau_F)/(t+2\tau_K) \sim 1.9,$$

from which one can calculate $\tau_F/\tau_K \sim 1.75$. These are the numbers obtained by me from numerous tests of the programs under discussion.

The sequence of the formulae composing this algorithm can easily be understood from the program given in the Appendix.

The general algorithm, applied to domains of arbitrary dimensions bounded with arbitrary surfaces (in the two-dimensional case we have lines), is described in Ref. 2, and one of its modifications will probably be published in the Soviet journal *Programming*.

The general algorithm has already proved its efficiency, for example in determining the location of the detector matter penetrated by a particle. This is very important when dealing with the huge number of particles handled by experimental high-energy physics.

Yours faithfully,

P. A. KALINCHENKO
USSR 142284, Moscow region, Protvino,
Institute for High Energy Physics,
Computer Centre

References

1. M. Smith, Points, polygons, and areas (letter to the Editor). *The Computer Journal*, **23** (2), 189 (1980).
2. P. A. Kalinchenko, Preprint, Institute for High Energy Physics **86-60**. Serpukhov (1986).

Appendix

```
      SUBROUTINE PPOLYN
     *  (X, Y, XP, YP, N, IV)
C     30 OCTOBER 1985
C     WRITTEN BY KALINCHENKO
      DIMENSION XP (N), YP (N)
      N1=N-1
      XPN=XP(N)
      YPN=YP(N)
      XN=X-XPN
      YN=Y-YPN
      IV=1
      P=1.
      ISTART=-1
      I=0
    1 IQ=-1
    2 I=I+1
      IF (I-N1)  4, 3, 16
    3 ISTART=-N
    4 XIN=XP(I)-XPN
      YIN=YP(I)-YPN
      IF ((XIN*YN-YIN*XN)*P)
     *  5, 7, 2
    5 P=-P
      IF (ISTART+1)  9, 2, 9
    7 IF (ISTART+I)  11, 11, 8
    8 IQ=1
    9 XIJ=XP(I)-XP(I-1)
      YIJ=YP(I))-YP(I-1)
      XJ=X-XP(I-1)
      YJ=Y-YP(I-1)
      IF ((XIJ*YJ-YIJ*XJ)*P)
     *  10, 12, 1
   10 IV=IQ*IV
      GO TO 1
   11 XIJ=XIN
      XJ=XN
      YJ=YN
   12 IF (XIJ)  13, 14, 13
   13 XI=X-XP(I)
      IF (XI*XJ)  (15, 15, 1
   14 YI=Y-YP(I)
      IF (YI*YJ)  15, 15, 1
   15 IV=0
   16 RETURN
C        -1-INSIDE
C    IV: 0-EDGE OR VERTEX
C          1-OUTSIDE
      END
```

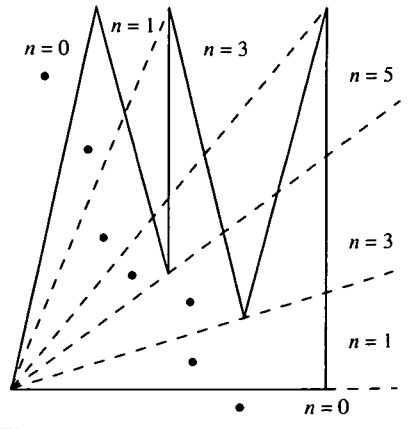


Fig. 1.