



**The University of Sydney**

**On Deflection Worm Routing on Meshes**

Technical Report Number 490

October 1994

Alan Roberts and Antonios Symvonis

ISBN 0 86758 943 4

**Basser Department of Computer Science  
University of Sydney NSW 2006**

# On Deflection Worm Routing on Meshes

**Alan Roberts   Antonios Symvonis**

Basser Department of Computer Science

University of Sydney

Sydney, N.S.W. 2006

Australia

{alanr,symvonis}@cs.su.oz.au

**Technical Report 490**

October 1994

## Abstract

In this paper, we consider the deflection worm routing problem on two dimensional  $n \times n$  meshes. In deflection routing a message cannot be queued and it is always moving until it reaches its destination. In worm routing, the message is considered to be a worm; a sequence of  $k$  flits which, during the routing, follow the head of the worm which knows the destination address. Our results include:

- (i) *An off-line algorithm for routing permutations in  $O(kn)$  steps.* Bar-Noy et al [1] described an  $O(kn)$ -step randomised algorithm for deflection worm routing of permutations on  $n \times n$  meshes. The  $O(kn)$  step randomised algorithm implies the existence of an off-line algorithm of the same complexity. We present an off-line algorithm which achieves this bound.
- (ii) *A general method to obtain deflection worm routing algorithms from packet routing algorithms.* We show how to derive a deflection worm routing algorithm from a packet routing algorithm which uses queues of size  $O(f(N))$  ( $N$  is the side-length of the mesh in which the packet routing algorithm is applied). Our result generalises the method of Newman and Schuster [3] in which only packet routing algorithms with a maximum queue of 4 packets can be used.

# 1 Introduction

Routing messages between the processors of a parallel machine is a crucial task which directly affects the performance of the machine. As a consequence, a huge amount of effort has been devoted to the development of efficient routing algorithms.

Usually, the parallel machine (or better, the underlying interconnection network) is represented as a directed graph where nodes represent the processors and directed edges represent unidirectional communication links. Some times undirected graphs are used as well, with the understanding that each undirected edge represents a bidirectional communication link. In the rest of the paper, we will assume that the processors operate in a synchronous mode.

Message routing has been abstracted in several ways. In *packet routing* it is assumed that a message can be transmitted between two adjacent processors in a single step as a *packet*. If packets can be stored in intermediate nodes during the trip from their origin to their destination, the routing model is referred as *store-and-forward*. In a different model known as *deflection* (or *hot-potato*) routing, packets continuously move between processors from the time they are injected into the network until the time they are consumed at their destination.

The advantage of deflection routing over the store-and-forward model is obvious. No queueing area is required at the processors. However, the fact that packets always move implies that at any step each processor must transmit the packets it received during the previous step (unless they were destined for it). As a result, several packets might be derouted away from their destination. This makes the analysis extremely difficult. Consequently, even though deflection routing algorithms have been around for several years [Baran64], most of the research focusses on the store-and-forward model.

The assumption that a whole message can be transmitted in a single step between adjacent processors is not a very realistic one, especially when the messages are long. It is more natural to assume that the amount of information that can be transmitted between processors in a single step is a hardware dependent variable (the *width* of the communication channel). This leads to the modelling of a message as a *worm*; a sequence of *flits*, each of size equal to the width of the communication channel, in which only the first flit knows the destination address. During the routing of a worm, all routing decisions are made by the processors which hold the head of the worm. The rest of the flits (the *body* of the worm) simply follow the path of the head. When the worms are allowed to be queued at intermediate processors waiting for the release of a communication link, we say that routing is performed according to the *store-and-forward worm routing* model. When queueing is not allowed, the *deflection worm routing* model is used.

In this paper, we will concentrate on deflection worm routing on  $n \times n$  meshes. Deflection worm routing on meshes was first examined by Bar-Noy, Schieber, Raghavan and Tamaki [1]. They studied permutation routing and presented  $O(k^{2.5}n2^{O(\sqrt{\log n \log \log n})})$ -step and  $O(kn^{1.5})$ -step deterministic and  $O(kn)$ -step randomised algorithms.

Newman and Schuster [3] described a method to obtain deflection worm routing algorithms based on store-and-forward packet routing algorithms. Their method was general enough to work for any routing patterns, not only permutations. However, the packet routing algorithms used in their method were restricted to use queues of at most four packets per processor. By employing the sorting algorithm of Schnorr and Shamir [4] they obtained an  $O(k^{2.5}n)$ -step deflection worm routing algorithm for routing permutations. They also presented an  $O(k^{1.5}n)$ -step off-line algorithm. Newman and Schuster also observed that better results for routing permutations could be obtained if fast algorithms for  $1 - h$  routing [2, 5] or  $h - h$  routing [5] were available. Sibyen and Kaufmann [5] used such algorithms to derive an  $O(k^{1.5}n)$ -step deflection worm routing algorithm for permutations.

This paper contributes to the literature of off-line and on-line deflection worm routing algorithms. We present an  $O(kn)$ -step off-line algorithm for routing permutations. The existence of such an algorithm

was implied by the  $O(kn)$ -step randomised algorithm of Bar-Noy et al [1] through standard but not constructive arguments. The best off-line algorithm known till now was the  $O(k^{1.5}n)$ -step algorithm of Newman and Schuster [3]. Sibyen and Kaufmann [5] managed to achieve the same number of steps by an on-line algorithm. Note that an  $O(kn)$ -step solution to a permutation problem is asymptotically optimal as a standard bisection argument reveals an  $\Omega(kn)$ -step lower bound. Finding an  $O(kn)$ -step off-line algorithm was posed as an open problem in [3].

Despite its effectiveness in deriving deflection worm routing algorithms from store-and-forward packet routing algorithms, the method of Newman and Schuster [3] has a major drawback. Only packet routing algorithms which use queues of at most 4 packets per node can be used. In this paper, we generalise their method to allow it to use packet routing algorithms of queue-size  $f(N)$ , where  $f(N)$  is a function of the side-length  $N$  of the mesh in which the packet routing algorithm is applied. The result dramatically increases the number of candidate packet routing algorithms that can be used in deriving deflection worm routing algorithms. Note also that the result also leads to simpler deflection worm routing algorithms since the packet routing algorithms which use queues of at most 4 packets are, in general, more complicated than those which use larger queues.

The rest of the paper is organised as follows: In Section 2, we present the  $O(kn)$ -step off-line algorithm. The algorithm is based on the *multistage off-line routing method* introduced by Symvonis and Tidswell [7]. We first review the method and then proceed with the algorithm and its analysis. In Section 3, we describe how to extend the method of Newman and Schuster [3] to utilise packet routing algorithms which use queues of size  $f(N)$ . We conclude in Section 4.

## 2 Optimal Off-line Deflection Worm Routing

In our effort to derive an off-line solution for routing permutations using the deflection worm routing model, we use the multistage off-line routing method [7]. The method was originally used for deriving off-line solutions to packet routing problems on meshes and tori. It was also used successfully in obtaining optimal off-line solutions for routing on trees [6]. In this section, we describe the multistage off-line routing method and adapt it to accommodate worm routing. We then describe the off-line routing algorithm.

### 2.1 The Multistage Off-line Routing Method

In its original definition, the multistage off-line routing method was able to treat the maximum allowable queue size as a parameter of the routing problem. Here, for simplicity (and since we are interested in deflection routing) we omit it from the specification of the routing problem.

A *finite directed graph*  $G = (V, E)$  is a structure which consists of a finite set of vertices  $V$  and a finite set of edges  $E = \{e_1, e_2, \dots, e_{|E|}\}$ . Each edge is incident to the elements of an ordered pair of vertices  $(u, v)$ .  $u$  is the start-vertex of the edge and  $v$  is its end-vertex. We will use the notation  $E(G)$  and  $V(G)$  to denote the edge set and the vertex set of graph  $G$ , respectively. A *directed path* is a sequence of edges  $e_1, e_2, \dots$  such that the end-vertex of  $e_{i-1}$  is the start-vertex of  $e_i$ . The set  $Neighbors(v, G)$  is defined to be the set of vertices in  $G$  which can be reached from  $v$  by crossing just one edge. Formally,  $Neighbors(v, G) = \{w \mid (v, w) \in E(G)\}$ .

Let  $M_n$  denote an  $n \times n$  two dimensional mesh.  $M_n$  has  $n^2$  vertices, each represented by an ordered pair of integers  $(row, column)$  where  $0 \leq row, column \leq n - 1$ . Each vertex  $(i, j)$ ,  $0 \leq i, j \leq n - 1$ , has outgoing edges to its four neighbors (provided that they exist)  $(i - 1, j)$ ,  $(i + 1, j)$ ,  $(i, j - 1)$ ,  $(i, j + 1)$ .

A *deflection packet (worm) routing problem*  $R$  is defined by a tuple  $(G, P)$  where  $G = (V, E)$  is the directed graph representing the network in which the routing will take place (vertices in  $V$  represent

processors and edges in  $E$  represent unidirectional communication links). The elements in set  $P$  represent the  $m$  packets (worms) to be routed. Formally,  $P = \{p_1, p_2, \dots, p_m \mid p_i = (orig_i, dest_i), orig_i, dest_i \in V, 1 \leq i \leq m\}$ .

There is no restriction on the number of packets (worms) which originate from, or, are destined for, a certain processor. If at most  $h_1$  packets (worms) originate from any processor, and, at most  $h_2$  packets (worms) are destined for any processor, then we say that we have an  $(h_1, h_2)$  routing problem. If  $h_1 = 1$  we have a *many-to-one routing problem* (many processors send packets (worms) to one processor), if  $h_2 = 1$  we have a *one-to-many routing problem* (one processor sends packets (worms) to many processors), and when  $h_1 = h_2 = 1$  we have the *permutation routing problem*.

Consider any routing problem  $R = (G, P)$  where  $G = (V, E)$  is the directed graph which represents the interconnection network in which the routing takes place and  $P$  is the set of packets to be routed. Our goal is to achieve routing time near the lower bound of the problem. Assume an upper bound of  $T$  routing steps for the problem under consideration (for now consider worst case trivial upper bound).

We construct a multistage directed graph  $G' = (V', E')$  as follows:

$$V' = \{(v, t) \mid v \in V \text{ and } 0 \leq t \leq T\}$$

and

$$E' = \{((v, t), (w, t + 1)) \mid w \in neighbors(v, G) \text{ and } 0 \leq t < T\}.$$

The edges in  $E'$  represent the communication that can take place between adjacent vertices of the interconnection network at any time.

Figure 1 shows the resulting graph when the interconnection network is a chain of length 5. For permutations on chains of processors, an obvious upper bound of 4 routing steps applies.

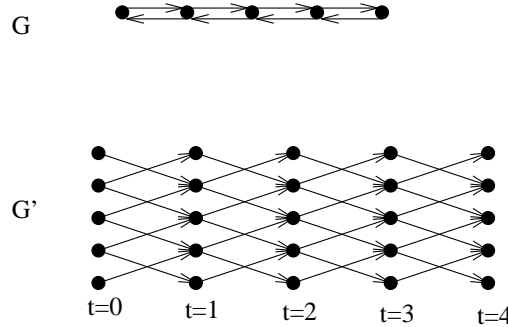


Figure 1: A chain of 5 vertices and its corresponding multistage graph.

Let  $tower(G', v)$  be the set of vertices of graph  $G'$  (the constructed multistage graph) which correspond to vertex  $v$  in  $G$ . Formally,

$$tower(G', v) = \{(v, t) \mid v \in V, (v, t) \in V', 0 \leq t \leq T\}.$$

We can think the stages of the multistage graph  $G'$  as representing time. In that sense, the route of any flit will be a directed path from a vertex in the flit's origin-tower to a vertex in the flit's destination-tower. So, an off-line solution to a deflection worm routing problem can be seen as a collection of paths. However, the paths must satisfy several conditions which reflect the fact that we are routing worms and that only one flit can be transmitted along any communication link in a single step.

**Definition** A valid off-line solution of length  $L$  for the deflection worm routing problem  $R = (G, P)$  is a set of directed paths, one path for each flit of each worm, in the multistage graph  $G'$  of  $G$ , such that:  
i) the head of worm  $p_i = (orig_i, dest_i) \in P$  travels from a vertex  $(orig_i, t')$  in  $tower(G', orig_i)$  to vertex

- ( $dest_i, t''$ ) in  $tower(G', dest_i)$ ,  $t' \leq t'' \leq L - k + 1$ , where  $k$  is the number of flits in a worm
- ii) if the  $j$ -th flit of a worm,  $1 \leq j < k$ , travels from vertex  $(v, t)$  to vertex  $(w, t + 1)$ , then the  $(j + 1)$ -th flit of the worm travels from vertex  $(v, t + 1)$  to vertex  $(w, t + 2)$ ,
  - iii) all paths are edge disjoint.

Given the above definition, the goal of an off-line deflection worm routing algorithm will be to derive a collection of paths, one for each flit of each worm, such that they form a valid off-line solution of the smallest possible length  $L$ .

## 2.2 The Off-line Algorithm

The off-line algorithm must describe for each worm the path that each of its flits takes. But, since we can deduce the movement of the whole worm from the movement of the head, only paths for the heads of the worms are necessary. Given a routing schedule of  $L$  steps and the fact that at most  $\frac{n^2}{k}$  worms might be moving at any step, we conclude that about  $\frac{Ln^2}{k}$  bits are necessary for the description of the routing schedule. Since  $L = \Omega(kn)$ , we expect that a routing schedule will require  $\Omega(n^3)$  bits for its description.

However, the routing schedule that our algorithm will produce can be described with  $O(n^2 \log(kn))$  bits. This is because the paths will be of a special form. More precisely, after each worm starts its routing it will move on a minimal path, first horizontally to its destination column and then vertically to its destination. Since these *one-bend* paths can be deduced from each (*origin, destination*) pair, the only information that is required to describe the routing of a worm is the step in which the worm starts moving. Thus,  $O(n^2 \log(kn))$  bits suffice.

In the description of the algorithm, variable  $start[p]$  contains the routing step in which worm  $p$  starts moving.

---

Algorithm *Off-line\_mesh\_routing*

1. Construct a multistage graph  $G'$  of  $4kn$  stages for an  $n \times n$  mesh as described in the previous section.
  2.  $G'_{current} = G'$
  3. **while** there are more worms to be routed **do**
    - (a) Let  $p = (orig, dest)$  be the next worm to be routed.
    - (b)  $stage = 1$
    - (c)  $routed = \mathbf{false}$  /\*  $routed$  will become true when a set of paths has been assigned to  $p$  \*/
    - (d) **while** (**not**  $routed$ ) **do**
      - i. Let  $S$  be the set of edges of  $G'$  which are required in order to route worm  $p$  in such a way that the head departs from node ( $orig, stage$ ) and moves horizontally to the column destination and then vertically to  $dest$ .
      - ii. **if**  $S \subseteq G'_{current}$  **then**

$$E(G'_{current}) = E(G'_{current}) - S$$

$$start[p] = stage$$

$$routed = \mathbf{true}$$
      - else**  $stage = stage + 1$
- 

**Definition** When algorithm *Off-line\_mesh\_routing* is used to produce a routing schedule, we say that worm  $p$  is *delayed* by worm  $q$  at step  $t$  if worm  $q$  used during its routing the first edge which prevents (because it was removed from  $G'_{current}$ ) worm  $p$  to be routed (as described in the algorithm) starting at time  $t$ .

**Lemma 1** Consider two worms  $p$  and  $q$  which are routed with algorithm `Off-line_mesh_routing`. *W.l.o.g.*, assume that paths have already been assigned to the flits of worm  $q$ . Then, during the path assignment phase (step 3(d)) of the algorithm, worm  $q$  can delay worm  $p$  by at most  $2k - 1$  steps.

**Proof** Worm  $p$  is delayed by worm  $q$  at time  $t$  if the first edge which is not present in  $G'_{current}$  but is required for the routing of  $p$  was used for the routing of a flit of  $q$ . The fact that the worms follow minimal paths to their destinations implies that a each flit of  $q$  can delay  $p$  at most once. We consider 2 cases:

- (i) An edge which was required for the routing of the head of  $p$  was used by  $q$ . This corresponds to the head of  $p$  “bumping into” worm  $q$ . Since worm  $q$  consists of  $k$  flits, it can delay  $p$  by at most  $k$  steps.
- (ii) An edge which was required for the routing of a flit of the body of  $p$  was used by  $q$ . The missing edge was used for the routing of the head of  $q$ . This corresponds to the head of  $q$  “bumping into” the body of  $p$ . Since the body of  $p$  consists of  $k - 1$  flits,  $p$  can be delayed by by at most  $k - 1$  steps.

From the two cases, we conclude that  $q$  can delay  $p$  by at most  $2k - 1$  steps. ■

**Theorem 1** Given an  $n \times n$  mesh and a permutation  $\pi$  of its vertices that has to be routed using the deflection worm routing model where each worm consists of  $k$  flits, Algorithm `Off-line_mesh_routing` produces an optimal routing schedule of  $O(kn)$ -steps.

**Proof** Consider an arbitrary worm  $p$ . When  $p$  is routed by algorithm `Off-line_mesh_routing`, it can interfere with at most  $2n - 2$  other worms, i.e.,  $n - 1$  worms which originate in the same row as  $p$  and  $n - 1$  worms which are destined for the same column as  $p$ . From Lemma 1 we know that each of these worms can delay  $p$  by at most  $2k - 1$  steps. Thus,  $p$  will start its routing after at most  $(2n - 2)(2k - 1) + 1$  steps. Since it might be at most  $2n - 2$  steps away from its destination,  $p$ 's tail will reach its destination after at most  $(2n - 2)(2k - 1) + 1 + (2n - 2 + k - 1) = O(kn)$  steps. A routing schedule of  $O(kn)$  steps is asymptotically optimal. ■

**Analysis of algorithm `Off-line_mesh_routing`** The algorithm requires memory to store the multistage graphs  $G'$ ,  $G'_{current}$ , set  $S$  and array `start`. This amounts to a total of  $O(kn^3)$ . A careful implementation of the algorithm can spare the use of  $G'$  and  $S$  (they were used in the presentation for clarity reasons). For a worst case time analysis assume that each worm is delayed for  $O(kn)$  steps. It takes  $O(kn)$  steps in the worst case to realize that the worm is being delayed. This is because the missing edge from  $G'_{current}$  might be one of the last edges in the worm's (almost successful) potential route. Thus, a route will be assigned to each worm after  $O(k^2n^2)$  time. Since there are  $n^2$  worms to be routed, the algorithm will terminate after  $O(k^2n^4)$  time. More careful implementation can reduce the time complexity to  $O(kn^4)$ .

**Variations of algorithm `Off-line_mesh_routing`** Several versions of algorithm `Off-line_mesh_routing` are possible and will produce similar results. What we presented is a very general version of it. We can refine the algorithm by specifying an order in which the packets will be routed. The arbitrary order used in the presentation resulted in an optimal algorithm. So any other ordering might reduce the constant factor hidden in the *big-Oh* notation. When  $k = 1$  the presented algorithm results in a routing schedule of  $4n - 4$  steps. If the worms (or packets, since they have only 1 flit) are routed in a lexicographical order with respect to the pair  $(hor, vert)$  where  $hor$  is the horizontal distance they have to travel and  $vert$  is the vertical one, a routing schedule of  $3n - 3$  steps will be produced (see [8] for details).

### 3 Routing on a Two-Dimensional Mesh

For the purposes of this section and in order to facilitate the task of drawing figures, the  $n \times n$  mesh  $M_n$  will be considered to be an undirected graph with edges capable of simultaneous transmission along both directions. We shall construct an efficient  $k$ -worm routing algorithm by treating each worm as though it were a packet and simulating the operation of a packet routing algorithm. Let  $\mathcal{A}(N)$  be the packet

routing algorithm (operating on  $M_N$ ) of which the operations we intent to simulate. We assume that  $\mathcal{A}(N)$  completes the routing within  $t_{\mathcal{A}}(N)$  steps and uses queues of size  $f(N)$  packets. Algorithm  $\mathcal{A}(N)$  is *suitable* for our method if the following assumption regarding the routing model is satisfied.

**Assumption 1** *On any given step all decisions made about the movement of any packet will be made locally by the node that currently stores the packets without considering the contents of any other node.*

In the original work of Newman and Schuster [3], any suitable for simulation packet routing algorithm  $\mathcal{A}(N)$  had to also satisfy:

**Assumption 2**  *$\mathcal{A}(N)$  uses a queue-size of at most 4 packets.*

Relaxing Assumption 2 results in an increase in the number of packet routing algorithms which are suitable for simulation. As we shall show, an  $O((f(N)k)^{2.5}n)$ -step algorithm will be derived if an  $O(N)$ -step packet routing algorithm which requires queue size of at most  $f(N)$  packets and satisfies Assumption 1 exists.

### 3.1 The Algorithm

Let  $\mathcal{A}(N)$  be a permutation packet routing algorithm which satisfies Assumption 1 and uses queues of size at most  $f(N) < n$  packets. Choose  $N$  such that satisfies  $N = n/(\sqrt{(f(N)+4)k} + 1)$ . We shall assume for simplicity that  $n, f(N)$  and  $k$  are integers such that  $N, \sqrt{k}$  and  $\sqrt{f(N)+4}$  are integers. In addition we will assume that  $k$  is even. Note that this immediately implies that  $\sqrt{(f(N)+4)k}$  is even<sup>1</sup>.

For the purposes of the algorithm, we divide the mesh up into an  $N \times N$  mesh of *supernodes*; each supernode being a sub-mesh of size  $(\sqrt{(f(N)+4)k} + 1) \times (\sqrt{(f(N)+4)k} + 1)$ . The rows and columns inside each supernode are numbered from 0 to  $\sqrt{(f(N)+4)k}$  in the same way that the overall mesh was.

The algorithm is then achieved in  $(\sqrt{(f(N)+4)k} + 1)^4$  *rounds*. During the  $(i, j, k, l)$ -th round we route all worms that originate at the  $(i, j)$  point of a supernode and are destined for point  $(k, l)$  of some supernode. Accordingly, at the beginning of each round, there is one worm generated inside each supernode. Each worm is then treated as though it were a packet. Decisions on sending worms from one supernode to another are made using a packet routing algorithm. The supernodes act like an  $N \times N$  mesh of nodes with respect to this packet routing algorithm. Each supernode has a queue size of at most  $f(N)$  packets. Each step of the packet routing algorithm is simulated by an underlying algorithm that determines the way in which the worms are moved about. From now on, we will refer to a step of the packet routing algorithm as a *superstep*. Each round proceeds “superstep” by “superstep” until all worms have arrived at their destinations. The algorithm proceeds “round” by “round” until the whole permutation has been routed.

The high level description of the algorithm is identical to that of Newman and Schuster [3]. However, since we allow the use of a larger class of packet routing algorithms, we have to modify the structure of the supernode and to drastically refine the simulation of the superstep.

---

<sup>1</sup>We can always satisfy these requirements by modifying  $k, f(N), n$  in a way such that the correctness of the algorithm and the asymptotic analysis are not affected.

### 3.2 The Structure of Each Supernode

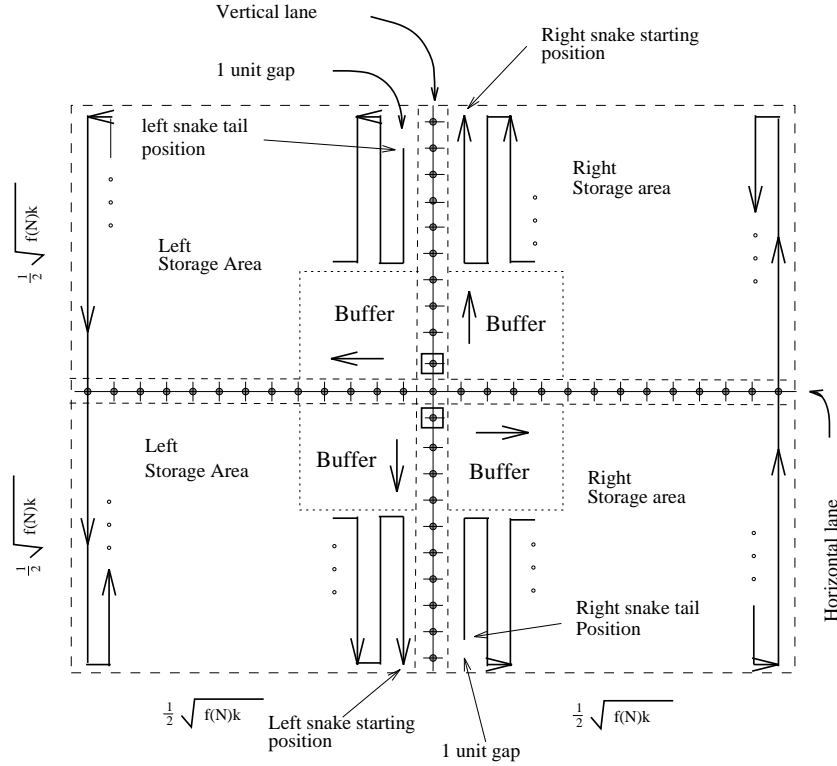


Figure 2: The sub-mesh that is used to simulate a supernode with respect to the packet routing algorithm  $\mathcal{A}(n)$ . Each buffer stores a single worm of length  $k$ .

The layout of the sub-meshes that are used to simulate supernodes participating in the packet routing algorithms is shown in Figure 2. Each sub-mesh contains several regions of importance. Below we give a basic description of their function by showing how they fit into the simulation of the packet routing algorithm. Later we will describe each of the phases of the simulation in detail.

Each sub-mesh is divided into quarters by the  $\frac{1}{2}\sqrt{(f(N) + 4)k}$  column and the  $\frac{1}{2}\sqrt{(f(N) + 4)k}$  row of processors. We shall refer to these as the vertical and horizontal *lanes*. There are two *command processors* positioned on the vertical lane. These processors are used to make the routing decisions of the supernode that is being simulated by the sub-mesh. The top command processor has coordinates  $(\frac{1}{2}\sqrt{(f(N) + 4)k}, \frac{1}{2}\sqrt{(f(N) + 4)k} - 1)$ , the bottom one has coordinates  $(\frac{1}{2}\sqrt{(f(N) + 4)k}, \frac{1}{2}\sqrt{(f(N) + 4)k} + 1)$ . During the simulation, both of these command processors are aware of the packet routing algorithm that is simulated and thus, if they both see the same data, they can make decisions which are influenced by their position in the sub-mesh but are not in conflict with the decision of the other command processor. The usefulness of this property will become evident during the description of the superstep simulation.

Within a supernode there are two main storage areas. One in each half of the sub-mesh. Within these areas worms are stored head to tail, forming a *snake*<sup>2</sup> that covers the storage area as shown in Figure 2. Each snake can store up to  $\frac{1}{2}f(N)$  worms. The snakes are used to simulate the queue of packets within a supernode.

There are several times during the simulation of a superstep when it is necessary for the command processors to consider all of the worms stored within a supernode. This is done by simultaneously moving the snakes of each storage area along a cycle that passes through the command processors. The cycle of the left hand storage area begins at the start position at the bottom left of the sub-mesh, as

<sup>2</sup>There must exist a better term. *Train* might be a more appropriate term, but, in that case, worms must be renamed to “waggons” and, unfortunately, waggons are not as flexible as worms....

shown in Figure 2. From there, it moves out up the vertical lane, through the command processors, and back into the storage area through the tail position shown. The cycle of the right hand storage area is the same except that it moves in the opposite direction, down the vertical lane. It takes exactly  $\frac{1}{2}f(N)k + \sqrt{(f(N) + 4)k}$  steps to complete one cycle. Note that because the algorithm is a hot-potato algorithm, the storage snakes will always be cycling around. The way that this will occur will be dealt with later when we discuss the different phases of the simulation in detail.

Transmission and reception of packets to and from neighbouring supernodes can occur on any given superstep of the packet routing algorithm. To facilitate this, each supernode contains four buffers. There is one buffer for each of the four directions that the simulated supernode can communicate in. Each buffer is designed to transmit in the direction of its arrow, as shown in figure 2. Each buffer is capable of holding a single worm which simulates the storage of a single packet. Transmission occurs along the vertical and horizontal lanes of processors. Prior to transmission, the worms which correspond to the packets that are to be transmitted on the next superstep of the routing, are selected from the storage areas and placed in the buffers. Once this has occurred, the worms then proceed from each buffer to the corresponding buffer in the neighbouring sub-meshes, using the vertical and horizontal lanes. After the outgoing worms have been transmitted from a supernode, incoming worms are stored in the buffers. The so-called *head processor* of each buffer is used to store the time that the buffer should transmit its worm.

In order to make the algorithm function with even  $k$ , it is necessary to make the communication phase take an even number of time steps. To do this it is necessary to have two types of buffers. Sub-meshes whose centre processor has an odd sum of coordinates will have *odd buffers* while sub-meshes whose centre processor has an even sum coordinates will have *even buffers* (Figure 3). As a consequence of this, it will also be necessary to keep the storage snake cycles of even supernodes<sup>3</sup> one step ahead of those of odd supernodes. Justification for these distinctions is given in the sub-section on transmission.

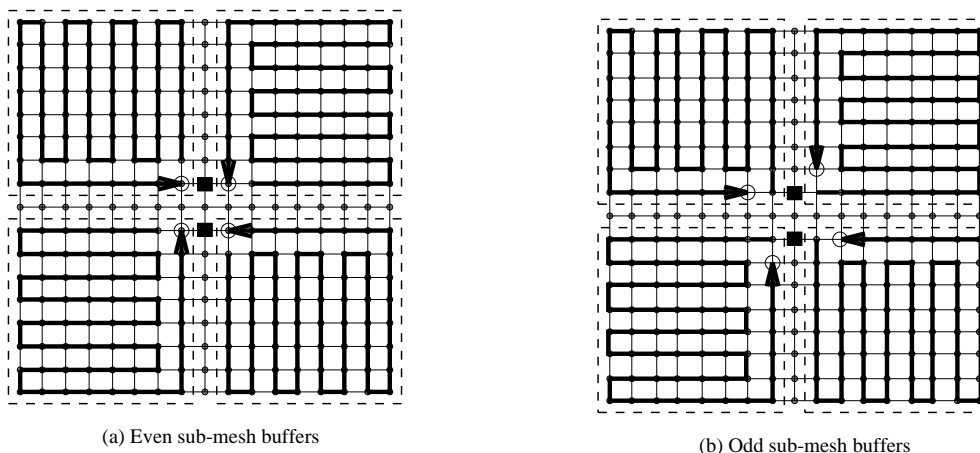


Figure 3: The buffers that are used to store incoming and outgoing worms. Figure (a) shows the buffers in an even sub-mesh. Figure (b) shows the buffers in an odd sub-mesh. The large white circle at the end of each worm head represent the head processor of each buffer.

### 3.3 The Simulation of a Superstep

Each superstep simulates the operation of a processor in the packet routing algorithm. An invariant of the simulation is that at the beginning of each superstep the buffers of a supernode are empty. This simply means that all packets currently at the supernode are held in the storage areas. Assuming that the precondition holds, a superstep can be considered to be a sequence of the phases:

<sup>3</sup>An *even (odd) supernode* is one with even (odd) buffer.

### *A Superstep*

- Output selection phase
- Extraction phase
- Transmission phase
- Queueing phase

At the beginning of each round, the worms that will be routed must be generated and placed in the storage snakes. This gives rise to another phase, the *creation phase* which we describe first. The description of a *consumption phase* at the end of each round is omitted since, by the time all the other phases are described, it will be obvious how to implement it.

### **Creation Phase**

Packets are created on the first step of a round. The creation of a packet within a supernode is simulated by the creation and positioning of a worm within a sub-mesh. At the beginning of a round all packets from the previous round have arrived at their destinations and have been absorbed. Accordingly, a sub-mesh is empty when a worm is created in it.

When a worm is created, it moves to the starting position (Figure 2) of the storage area that is nearest its birth place. In doing this it becomes the first worm of a snake for that storage area. Since all worms are born in the same place within each sub-mesh, all worms will reach the starting position at the same time. In the next phase these new-born worms will be placed in the buffers in order to prepare for transmission. In ‘even’ sub-meshes this will take one step longer than it will in odd sub-meshes. In order to synchronise transmissions from all sub-meshes we therefore create worms in even sub-meshes one step earlier than we do in odd sub-meshes. The consequence of this is that the snake cycles of ‘even’ sub-meshes will be one step ahead of those of odd sub-meshes. This will remain an invariant during all of the phases of the simulation and in fact, throughout the entire algorithm.

The distance that new-born worms have to travel to reach their prescribed starting position is  $O(k + \sqrt{(f(N) + 4)k})$ . The time cost for this phase is therefore  $O(k + \sqrt{f(N)k})$ .

### **Output Selection Phase**

At the beginning of any given superstep, a supernode may have several packets in its queue. The supernode decides which (if any) of these to transmit to neighbouring supernodes. This process is simulated by selecting worms from the storage snakes of a sub-mesh. In order to select which worms to output during the current superstep of the routing, the command processors must review all of the worms within a sub-mesh. This is done by moving the snake of each storage area along a cycle that passes through the command processors, as described previously.

Once this first cycle is complete, all worms will have passed through the command processors. By this time the command processors will have decided which worms (if any) are to be transmitted on the next step of the packet routing algorithm. In the queueing phase which occurs later, it will be necessary to place incoming packets into the queues of each supernode. This will be simulated by placing incoming worms into the storage snakes. To do this in a satisfactory way it is necessary to guarantee that there are at least two worm-sized gaps in each storage snake after the output selection phase has ended. This is accomplished during a second cycle of the storage snakes.

Assumption 1 implies that on any superstep each supernode must have room for at least four new packets. A direct consequence of this is that there will always be at least four worm-sized gaps somewhere

in the storage snakes of a sub-mesh. If there are less than two gaps in one particular snake then there will be more than two in the other one. Consider the processor that is at the centre of the sub-mesh. As the snakes cycle around they pass through it in opposite directions. Eventually there will be a gap on one side of it but not on the other. When this occurs the worm that is entering the processor from one side is reflected backwards, filling the gap that is entering the processor from the other side. In this way we take gaps away from one snake and put them into the other one. This process continues until both snakes have at least two gaps in them. This is certain to be the case once this second cycle of the storage snakes is over.

When incoming worms are inserted into the snakes during the queueing phase, it will be necessary for the head processors of the buffers (Figure 3) to know when to send their worms out into the vertical lane in order to fit them into a gap. As soon as both snakes have at least two gaps in them, the command processors assign two of the gaps in the left snake to the bottom buffers, one each. In the same way, two of the gaps in the right snake are assigned to the top buffers, one each. Information about each of the assigned gaps is then sent to each buffer by the command processor closest to it.

The output selection phases ends once the second cycle of the storage snakes is complete. The total time taken to complete output selection is therefore  $O(f(N)k)$  steps.

### Extraction Phase

Once the output selection phase is complete, the chosen worms must be extracted into the buffers in order to transmit them. This is accomplished during the extraction phase. The storage snakes cycle around again and the chosen worms are extracted into the buffers by the command processors. Worms that have to go to either of the top two buffers will be extracted by the top command processor. Worms that have to go to either of the bottom two buffers will be extracted by the bottom command processor. Extraction of a worm into an even buffer (see figure 3) takes  $k + 1$  steps, whereas extraction into an odd buffer takes  $k$  steps. The one step difference between the snake cycles of even and odd sub-meshes will ensure that this difference in buffering times does not cause a synchronisation problem.

After a worm has been extracted from a snake, the snake continues to cycle around. It may be some time before all of the worms have been extracted and transmission is ready to occur. In order to preserve the hot-potato property of the algorithm, extracted worms are therefore made to cycle inside the buffers until transmission is ready to occur. To complete a buffer-cycle, the worm's head simply follows the worm's tail around until the worm is back at its starting position again. Each such cycle takes exactly  $k$  steps.

Observe that if we simply extract the worms directly from the snakes then there will be a problem. The problem is that any worm extracted from a snake into one of the bottom buffers will be two steps out of phase with a worm extracted from the same snake into one of the top buffers. This problem is solved as follows. When a worm is extracted from the left snake into one of the bottom buffers, an additional delay of two steps is added using the method shown in Figure 5. Similarly, when a worm is extracted from the right snake into one of the top buffers an additional delay of two steps is added. In this way all buffer cycles are synchronised.

The extraction phase ends when transmission is ready to occur. Transmission from the top buffers cannot occur until the tail of the left storage snake is at or above the top command processor. Transmission also cannot occur unless the buffered worms are in their starting position, as shown in Figure 3. Figure 4 shows the positions of the snake tails when transmission is ready to occur from the top buffers of a sub-mesh. The figure shows both the "odd" and "even" cases. Note that the bottom buffers of a sub-mesh transmit at the same time as the top ones. The snakes of a sub-mesh always reach the position shown in Figure 4, before completing their second cycle. The extraction phase is therefore completed in  $O(f(N)k)$  steps.

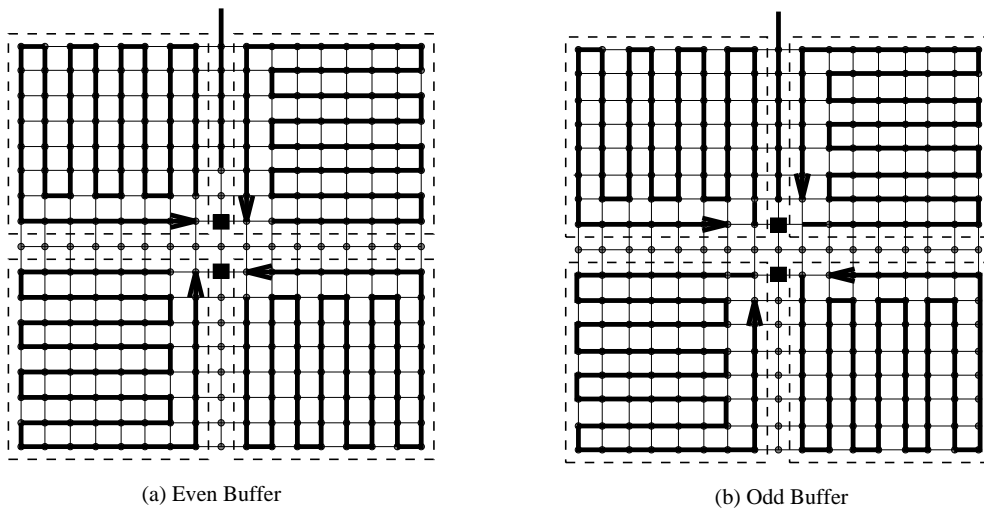


Figure 4: The position of the storage snake tail when the top buffers are ready to transmit. Figure (a) shows the “even” case. Figure (b) shows the “odd” case.

### Transmission

Once the extraction phase has ended, the chosen packets are transmitted from each supernode to the neighbouring supernodes. To simulate this, the worms contained in the buffers simultaneously move down the lanes that are anti-clockwise adjacent to them, in the directions shown in Figure 2. Each worm continues to move down its lane until it reaches the entry point of the first buffer on its right. It then moves into this buffer.

As promised earlier, we now explain why it is necessary to have odd and even buffers. On any superstep it may be necessary to keep some packets queued up in a supernode until the next superstep. As explained previously this is simulated by cycling the storage snakes.  $k$  is even. It is therefore necessary to make the travel time between sub-meshes take an even number of steps. If we did not, the transmitted worms would be out of phase with the storage snakes of their target sub-mesh and it might then become impossible to insert them for storage.

If a worm originates in an odd buffer, its destination buffer will be even. As it stands, it will take such a worm  $k + \sqrt{f(N)}k - 2$  steps to reach the destination buffer and be stored in there. On the other hand, if a worm originates in an even buffer, its destination buffer will be odd. In that case, transmission will take  $k + \sqrt{f(N)}k$  steps.

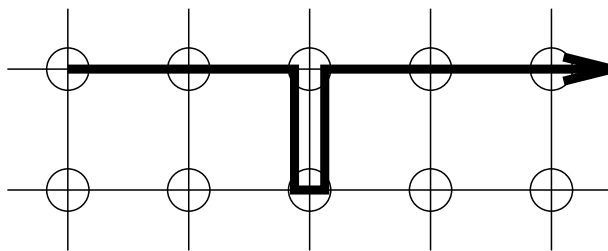


Figure 5: A 2 step delay. Many such delays can be executed, one for each processor along the path of a worm. In this way it is possible to produce even delays of any length, limited only by the length of the path.

The algorithm is a hot-potato algorithm. Accordingly, during the time that this communication is taking place, it is necessary to keep the storage snakes of each sub-mesh moving. It is also necessary to make sure that the storage snakes do not block any worms that are being transmitted along the vertical

lanes. It is therefore necessary to add a delay to the extraction cycle using the method shown in Figure 5. Note that the addition of this delay will not affect the extraction phase in any way.

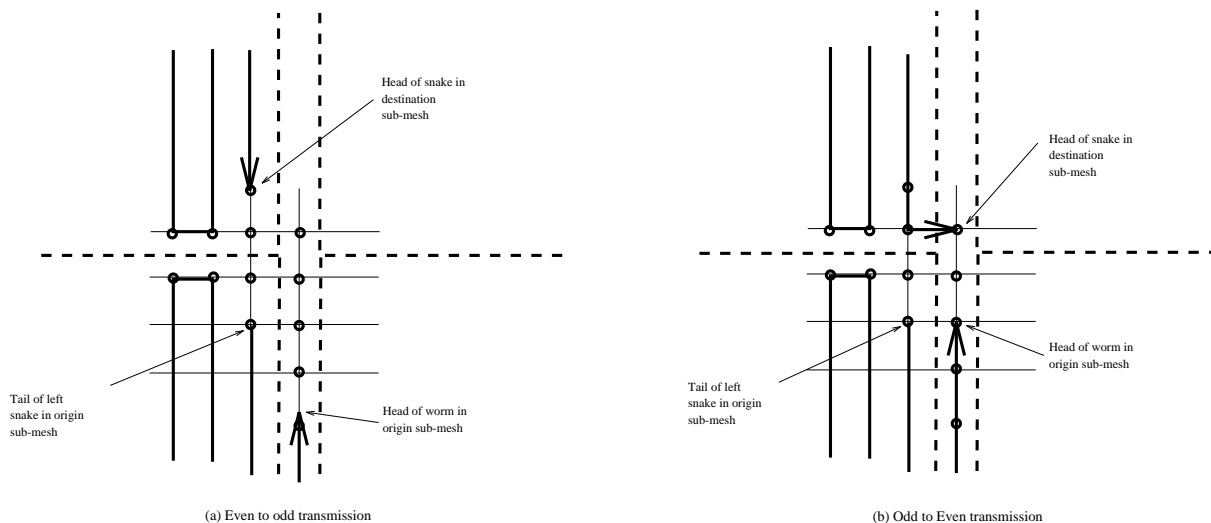


Figure 6: The snake positions in origin and destination sub-meshes relative to a worm moving up the vertical lane. Figure (a) shows the “even to odd” case. Figure (b) shows the “odd to even” case. As can be seen, a delay must be added to the snake cycles in order to prevent a collision.

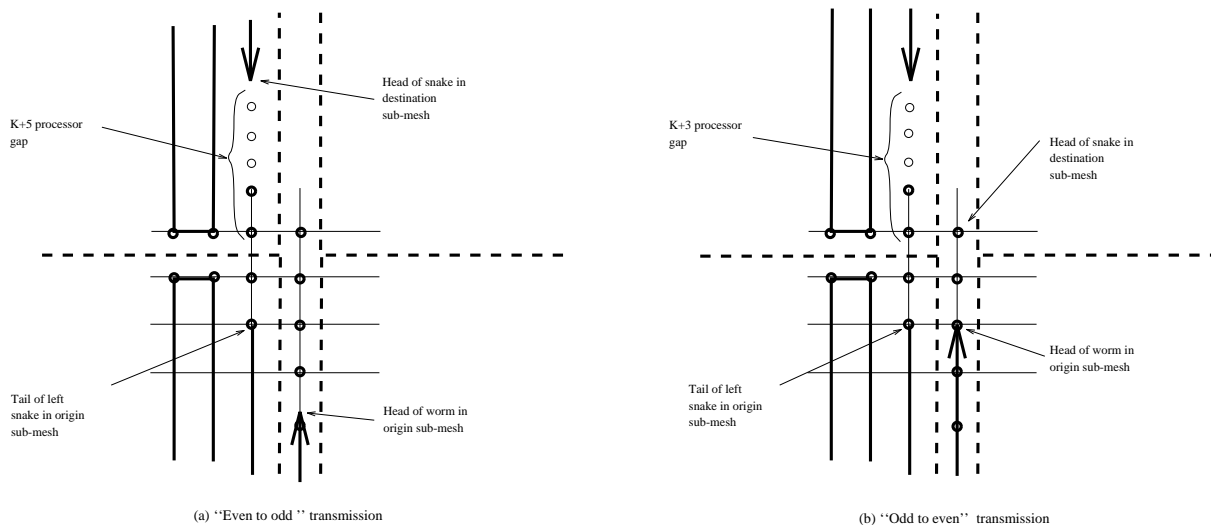


Figure 7: The snake positions in origin and destination sub-meshes relative to a worm moving up the vertical lane, after the appropriate number of delay-steps have been added to guarantee that no collision occurs during transmission. Figure (a) shows the “even to odd” case. Figure (b) shows the “odd to even” case.

As we have just found out, it takes  $k + \sqrt{f(N)k}$  steps to complete transmission of worms from even to odd sub-meshes. It takes  $k + \sqrt{f(N)k} - 2$  steps for transmissions from odd to even sub-meshes. When a worm is about to cross from one sub-mesh into another via the vertical lane, the situation will be as shown in Figure 6. In order to prevent the blocking of transmission and ensure that the snakes are in the correct position to insert incoming worms once transmission has occurred, a delay of  $k + 4$  steps is added to the extraction cycle of all sub-meshes. To compensate for the difference between odd and even sub-meshes, a two step delay is added during the buffering of incoming worms in even sub-meshes. Figure 7 shows the situation of Figure 6, after these delays have been added.

## Queueing

Once the transmitted worms have arrived in a supernode, they must be stored in its queue. This is simulated by the insertion of the received worms from the buffers into the storage snakes. In the output selection phase we guaranteed that each storage snake would have at least two worm-sized gaps in it. It is into these gaps that we now insert the transmitted worms.

During the output selection phase, one gap was assigned to each buffer. The storage snakes now come around for another cycle. As the gaps in the snakes pass by the buffers, the worms are inserted into the gaps via the command processors. Any one of these buffered worms may have to wait for some time before the gap assigned to its buffer has come around. Accordingly, worms cycle inside their buffers until their gap becomes available. Any worms that are in the bottom two buffers are inserted into gaps in the left storage snake. Any worms that are in the top two buffers are inserted into gaps in the right storage snake. Due to the actions taken in the output selection phase, each snake is guaranteed to have at least two gaps in it, ensuring that there will always be enough gaps available to do insertions in this way. In order to insert a worm into one of the storage snakes, its gap must reach the command processor at the exact same time that it does. The delays mentioned in the previous sub-section ensure that this will occur. All worms are certain to have been inserted into storage once the snakes have completed their queueing cycle. The queueing phase is therefore completed in  $O(f(N)k)$  steps.

From the above discussion, it is obvious that a superstep can be simulated in  $O(f(N)k)$  steps of the deflection worm routing algorithm. This leads to the following theorem:

**Theorem 2** *Let  $\mathcal{A}(N)$  be an  $O(N)$ -step permutation packet routing algorithm for  $M_N$  which uses queues of size  $f(N)$  and satisfies Assumption 1. Then there is a hot-potato worm permutation routing algorithm  $\widehat{\mathcal{A}}(n)$  for  $M_n$ , which routes  $k$ -worms in  $O((f(N)k)^{2.5}n)$  steps, where  $N$  satisfies  $N = \frac{n}{\sqrt{(f(N)+4)k+1}}$ .*

**Proof** The algorithm presented in the previous section requires  $(\sqrt{f(N)k} + 1)^4$  rounds to complete. In each round, a complete run of the packet routing algorithm  $\mathcal{A}(N)$  is simulated on an  $N \times N$  mesh of supernodes where each supernode is simulated by a sub-mesh. Each superstep of the routing that takes place during a round is simulated by several different phases. As we have seen, no phase requires more than  $O(f(N)k)$  steps to complete. Each round is therefore completed in  $O(f(N)kN)$  steps. The whole algorithm therefore takes  $O((f(N)k)^{2.5}n)$  steps to complete. Since the algorithm can route any partial permutation, we conclude that the theorem is true. ■

**Note:** An upper bound on the worm-size must be placed in order for Theorem 2 to be valid. It can be easily seen that  $1 \leq k \leq \alpha \frac{n^2}{f(N)}$ , for a small positive constant  $\alpha$ , must hold.

### 3.4 Many-to-One Routing

Consider the assumption that was made about  $\mathcal{A}(n)$ . Assumption 1 does not restrict it to permutation routing. It is equally valid to use the same simulation to perform other kinds of routing. However, special consideration must be taken for the generation of the worms at the beginning of each round and for the consumption of worms that reach their destination supernode. Consuming worms that arrive to their destination supernode must be done at the end of each superstep. In the following, we describe the *consumption phase* which can be added as the last phase of the superstep simulation to handle the general case.

## Consumption Phase

Once the queueing phase has been completed, it is possible that some worms may have reached their destination supernodes. Such worms must be consumed. The worm's target processor may lie anywhere within the sub-mesh. At the end of the queueing phase the buffers are empty and all worms are stored inside the storage snakes. The snakes then begin a *death cycle* as shown in Figure 8. If a worm has a target processor that lies on the path of the cycle, it will be absorbed as the cycle goes around. Worms that have targets which lie in the buffers or the lanes, will be sent there by the appropriate white circled processors. Each white circled processor is responsible for sending worms to a particular region. There is one white circled processor for each buffer and two for each lane, one at each end. If a worm has to go to one of these places, it will be drawn out of the cycle by the first white circled processor it meets that is assigned to its target region, as the cycle passes through.

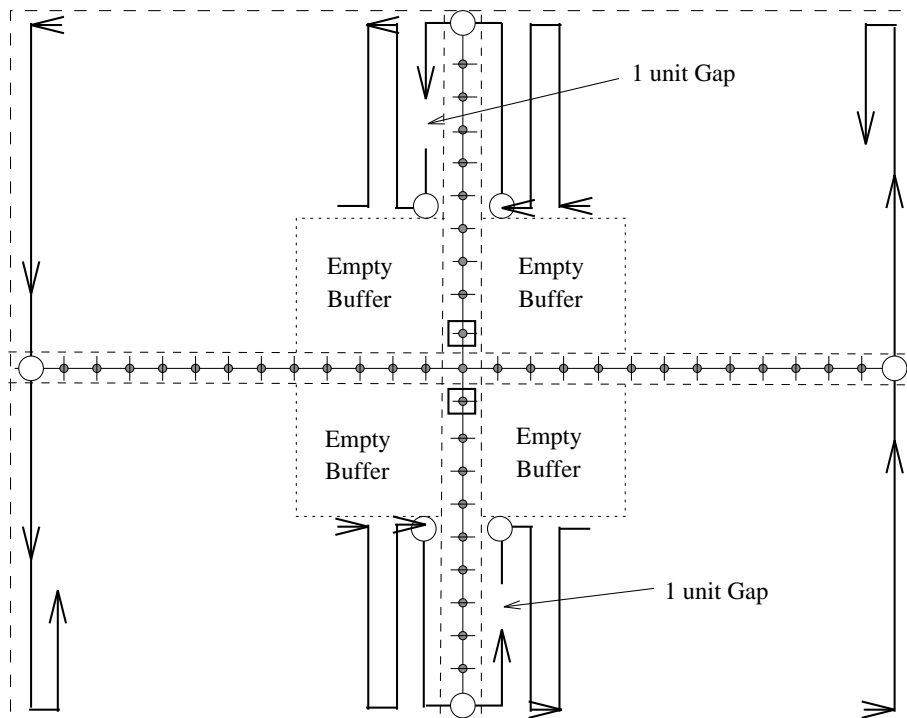


Figure 8: The death cycle of the worms within a sub-mesh. The cycle passes completely through both storage areas. Any worm that has a target processor in the sub-mesh which does not lie on the cycle will be sent to its target by the appropriate white circled processor.

The death cycle takes  $f(N)k + 4$  steps to complete. At the end of the death cycle it is possible that the absorption of some worms may not have finished yet. To take account of this fact and to ensure synchronisation, the storage snakes must be delayed by a small number of extra steps as described in Figure 5. We conclude that the absorption phase takes  $O(f(N)k)$  steps to complete.

## 4 Conclusions

In this paper, we described an algorithm for off-line deflection worm routing of permutations in an  $n \times n$  mesh in  $O(kn)$ -steps, where  $k$  is the number of flits in each worm. This result suggests that it might be possible to get the same (optimal) performance with an on-line algorithm. Designing such an algorithm is an open problem. We also generalised the method of Newman and Schuster [3] for obtaining deflection worm routing algorithms on two-dimensional meshes by simulating known packet routing algorithms.

Our contribution is the relaxation of the requirement that the simulated packet routing algorithm uses queues of at most 4 packet.

## References

- [1] A. Bar-Noy, B. Schieber, P. Raghavan, and H. Tamaki. Fast deflection routing for packets and worms. In *Proceeding of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC 93)*, Ithaca, NY, pages 75–86, August 1993.
- [2] F. Makedon and A. Symvonis. Optimal algorithms for the many-to-one routing problem on two-dimensional meshes. *Microprocessors and Microsystems*, 17(6):361–367, 1993.
- [3] I. Newman and A. Schuster. Hot potato worm routing via store-and-forward packet routing. In *First Israel Symposium on Theory of Computing and Systems (ISTCS'93)*, 1993. To appear in *Journal of Parallel and Distributed Computing*.
- [4] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the 18th Ann. ACM Symposium on Theory of Computing (Berkeley, CA)*, pages 255–263. ACM, ACM Press, 1986.
- [5] J.F. Sibeyn and M. Kaufmann. Deterministic  $1 - k$  routing on meshes. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science, STACS 94 (Caen, France, February 1994)*, LNCS 775, pages 237–248. Springer-Verlag, 1994.
- [6] A. Symvonis. Optimal algorithms for packet routing on trees. In *Proceedings of the 6<sup>th</sup> International Conference on Computing and Information (ICCI'94)*, Peterborough, Ontario, Canada, pages 144–161, May 1994. Also TR 471, Basser Dept of Computer Science, University of Sydney, September 1993.
- [7] A. Symvonis and J. Tidswell. A new approach to off-line packet routing. Case study: 2-dimensional meshes. In *Proceedings of the 1992 DAGS/PC Symposium Dartmouth Institute for Advanced Graduate Studies in Parallel Computation, Hanover, NH, USA*, pages 84–93, June 1992.
- [8] A. Symvonis and J. Tidswell. An empirical study of off-line permutation packet routing on 2-dimensional meshes based on the multistage routing method. Technical Report TR-477, Basser Dept of Computer Science, University of Sydney, February 1994. Submitted for journal publication.