A General Method for Deflection Worm Routing on Meshes Based on Packet Routing Algorithms

Alan Roberts, Antonios Symvonis,

Abstract— **In this paper, we consider the deflection worm routing problem on $n \times n$ 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. 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 in which only packet routing algorithms with a maximum queue of 4 packets can be used.**

Keywords— **Deflection routing, Mesh connected computer, On-line routing algorithm, Packet routing, Permutation routing, Worm routing,**

I. 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 assume that the processors operate in a synchronous mode and that they are able to simultaneously transmit and receive along all of their communication links.

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 some packets reach the processor they are destined for). 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 [2], most of the research focuses 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 assume that each worm consists of k flits, that is, all worms are of equal size.

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*. When $h_1 = 1$ and $h_2 > 1$ we have a *many-to-one routing problem* (*many* processors send packets (worms) to *one* processor), when $h_1 > 1$ and $h_2 = 1$ 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*.

In the limited space available, it is almost impossible to provide a complete set of references on different routing models and on routing algorithms for these models on different interconnection networks. For this reason, we only provide references that are necessary for the development/understanding of the results in this paper. We refer readers interested in the area of routing to Leighton's book [6], to Tompa's lecture notes [16] and to the survey article on methods for message routing [7].

In this paper, we 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 $O(k^{2.5} n 2^{O(\sqrt{\log n \log\log n})})$-step and $O(kn^{1.5})$-step deterministic and $O(kn)$-step randomised algorithms. Newman and Schuster [10] 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 [13] 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 [9], [14] or $h-h$ routing [14] were available. Sibyen and Kaufmann [14] used such algorithms to derive an $O(k^{1.5} n)$-step deterministic deflection worm routing algorithm for permutations. Finally, Roberts and Symvonis [12] developed an $O(kn)$-step deterministic off-line algorithm based on the *multistage off-line routing method* [15].

Even though the method of Newman and Schuster [10] is effective in deriving deflection worm routing algorithms from store-and-forward packet routing algorithms, it has a major

A. Roberts and A. Symvonis are with the Basser Department of Computer Science, University of Sydney, Sydney, N.S.W. 2006, Australia. Email:{alanr,symvonis}@cs.su.oz.au.

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 use of packet routing algorithms of queue-size $f(N)$ results to worm routing algorithms for k-flit worms on $n \times n$ meshes which terminate after $O((f(N)k)^{2.5}n)$ steps, where N satisfies $N = \frac{n}{\sqrt{(f(N)+4)k}+1}$ and $1 \leq k \leq \alpha \frac{n^2}{f(N)}$, for a small positive constant α.

Despite being a theoretical result which proves that the restriction placed on the queue size of the simulated packet routing algorithm is unnecessary, the result increases the number of candidate packet routing algorithms that can be used in deriving deflection worm routing algorithms (for example, the algorithms in [4], [5], [8], [9], [11]) and 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 paper is organised as follows: In Section 2, we give a high-level description of the routing algorithm. The algorithm partitions the $n \times n$ mesh into groups of processors that are referred as *supernodes*. In Section 3, we describe the operational structure of each supernode, i.e., the special role that each processor of the supernode plays during the routing. In Section 4, we describe how a single routing step of the packet routing algorithm is simulated by the processors of the $n \times n$ mesh which operate in the worm deflection routing model. We conclude in Section 5.

II. A High-Level Description of the Routing Algorithm

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 construct an efficient k-worm routing algorithm by treating each worm as though it were a packet and by simulating the operations 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 regarding the movement of any packet are made locally by the node that currently stores the packet without considering the contents of any other node.

In the original work of Newman and Schuster [10], 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 show, an $O((f(N)k)^{2.5}n)$-step algorithm can be derived from an $O(N)$-step packet routing algorithm which requires queue of size at most $f(N)$ packets and satisfies Assumption 1.

A. The Algorithm

For simplicity of exposition, we consider only the case where we have to route a permutation. As it is pointed out in the closing section of the paper, the algorithm is able to handle any routing pattern, provided that a suitable packet routing algorithm exists.

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 it satisfies $N = n/(\sqrt{(f(N)+4)k}+1)$. Without loss of generality, we assume that n, N, $f(N)$, $\sqrt{f(N)+4}$, k, and $\sqrt{k}$ are integers. In addition, we assume that k is even. Note that this immediately implies that $\sqrt{(f(N)+4)k}$ is even[1].

For the purposes of the algorithm, we treat the $n \times n$ mesh as 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 is.

The algorithm then consists of $(\sqrt{(f(N)+4)k}+1)^4$ *rounds*. During the (i,j,k,l)-th round, $0 \leq i,j,k,l \leq \sqrt{(f(N)+4)k}+1$, we route all worms that originate at node (i,j) of a supernode and are destined for node (k,l) of some supernode. Accordingly, since we consider permutations, at the beginning of each round, there is at most one worm generated inside each supernode and at most one worm is destined for every supernode. Each worm is 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 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 [10]. 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.

III. The Operational Structure of a Supernode

The layout of the sub-meshes that are used to simulate supernodes participating in the packet routing algorithms is shown in Figure 1. Each sub-mesh contains several regions of importance. Below we give a description of their function by showing how they fit into the simulation of the packet routing algorithm. In the next section, we 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 refer to these as the vertical and horizontal *lanes*. There are two *command processors* positioned on the vertical lane. These processors 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

[1]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.

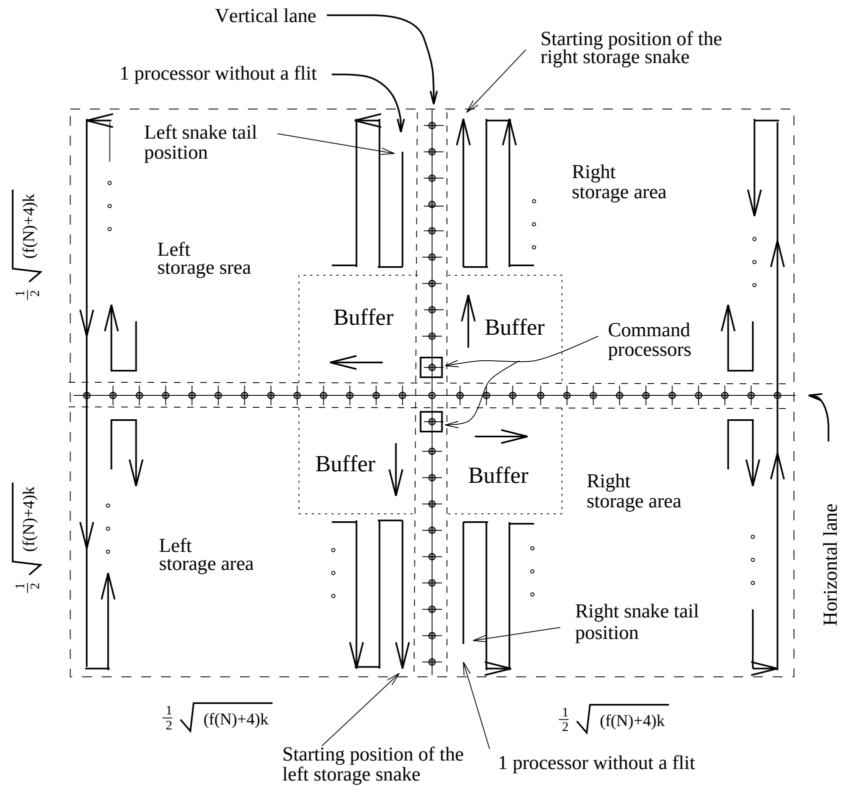

Vertical lane
1 processor without a flit
Starting position of the right storage snake
Left snake tail position
Left storage srea
Right storage area
Buffer
Buffer
Command processors
Buffer
Buffer
Right storage area
Left storage area
Horizontal lane
Right snake tail position
$\frac{1}{2}\sqrt{(f(N)+4)k}$
Starting position of the left storage snake
1 processor without a flit

Fig. 1. 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.

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*[2] that covers the storage area as shown in Figure 1. Each snake can store up to $\frac{1}{2}f(N)$ worms. The snakes simulate the queue of packets within a supernode.

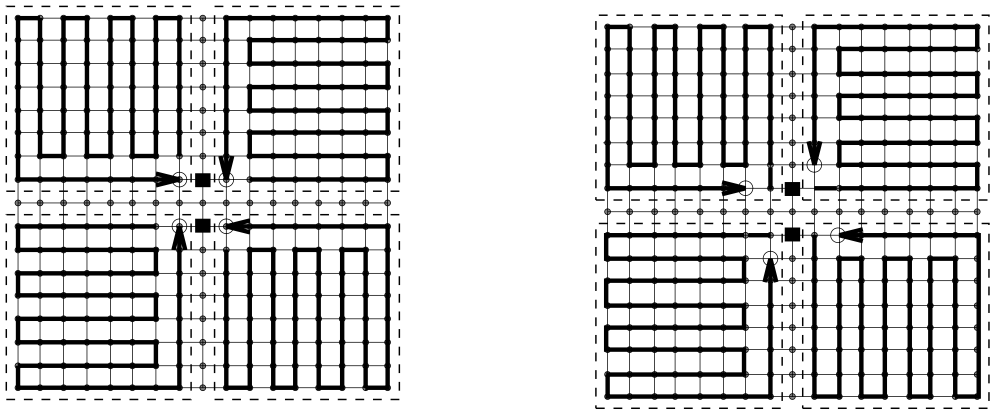
There are several times during the simulation of a superstep that it is necessary for the command processors to consider all of the worms stored within a supernode. This is done by simultaneously moving the snakes of each storage area along a cycle that passes through the command processors. The cycle of the left storage snake begins at its start position at the bottom row of the sub-mesh, as shown in Figure 1. 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 storage snake 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 deflection algorithm, the storage snakes are always cycling around (we show how this is done in the next section). At various times during the execution of the algorithm the storage snakes may contain gaps or may even be empty. However, for the purposes of timing the different phases we always consider the head and tail positions of the storage snakes to be where they would be if the snake were full.

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.* Each buffer is capable of storing a single worm of k flits and 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 1. 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. When the time comes for a worm to leave its buffer it does so by taking a right hand turn at the head processor of the buffer and then proceeding in the direction of its arrow, as shown in Figure 1.

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 achieve this, it is necessary to have two types of buffers. Sub-meshes whose center processor has an odd sum of coordinates have *odd buffers* while sub-meshes whose center processor has an even sum coordinates have *even buffers.* See Figure 2. (Notice also the position of the head processor in odd and even buffers.) As a consequence of this, it is also necessary to keep the storage snake cycles of even supernodes[3] one step ahead of those of odd supernodes. Justification for these distinctions is given when the transmission of a worm from a supernode is described.

[2]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....

[3]An *even (odd) supernode* is one with an even (odd) buffer.


(a) Even sub-mesh buffers

(b) Odd sub-mesh buffers

Fig. 2. 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's head represents the head processor of each buffer.

IV. The Simulation of a Superstep

Each superstep simulates the operation of a processor (i.e., a supernode) during a single step of 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 following phases:

A Superstep

- *Output selection phase:* We select from the stored worms the ones to be transmitted.
- *Extraction phase:* We extract the selected worms from the storage snakes, and we place them in the appropriate buffers.
- *Transmission phase:* We transmit the buffered worms to the neighbouring sub-meshes where they are, again, temporarily buffered.
- *Queueing phase:* We store the worms that have just arrived during the transmission phase and place them into the storage snakes.
- *Consumption phase:* We consume the worms that have arrived at their destination sub-meshes (if any).

At the beginning of each round, we first generate, and then we place in the storage snakes the worms that are to be routed. This gives rise to another phase, the *creation phase* which we describe first.

A. 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 the corresponding 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 1) of the storage area that is nearest its *birth place* (origin processor). In doing so, it becomes the first worm of a snake for that storage area. Since all worms are created in the same processor within each sub-mesh, all worms reach the starting position at the same time. In the next phase these newly created worms are placed in the buffers in order to prepare for transmission. In even sub-meshes this takes one step longer than 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 are one step ahead of those of odd sub-meshes. This remains an invariant during all of the phases of the simulation and in fact, throughout the entire algorithm.

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

B. 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 decide which worms to transmit 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 cycle is completed, all worms have passed through the command processors. By this time the command processors have decided which worms (if any) are to be transmitted on the next step of the simulated packet routing algorithm.

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

C. 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 are extracted by the top command processor. Worms that have to go to either of the bottom two buffers are extracted by the bottom command processor. When a worm is extracted from a vertical lane into a buffer on its right (left), it takes a right (left) turn at the relevant command processor and proceeds directly to its right (left) curling itself up into the buffer. Extraction of a worm into an even buffer (see Figure 2) takes $k + 1$ steps, whereas extraction into an odd buffer takes k steps. Since the snakes cycles of even sub-meshes are one step ahead of the odd sub-meshes, this ensures that the 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 deflection 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, there is a problem. Any worm extracted from a snake and moved to one of the bottom buffers is two steps out of phase with a worm extracted from the same snake and moved to one of the top buffers. This problem is solved as follows: When a worm is extracted from the left snake and moved to one of the bottom buffers, an additional delay of two steps is added using the method delineated in Figure 3. Similarly, when a worm is extracted from the right snake and moved to one of the top buffers an additional delay of two steps is added. In this way all buffer cycles are synchronised.

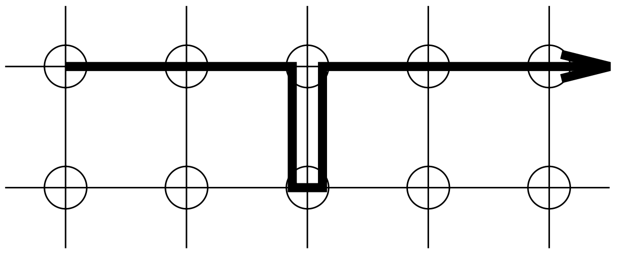


Fig. 3. A 2 step delay. Many such delays can be executed, one for each processor along the path of a worm. In this way it is possible to produce even delays of any length, limited only by the length of the path.

In the queueing phase which occurs later, it is necessary to place incoming packets into the queues of each supernode. This is 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 extraction phase has ended. This is accomplished by having the storage snakes cycle another time in this phase.

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 always are 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 are more than two in the other one. Consider the processor that is at the center of the sub-mesh. As the snakes cycle around, they pass through it in opposite directions. Eventually there is 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 is necessary for the head processors of the buffers (Figure 2) 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 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 2. 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 the second cycle of the extraction phase. The extraction phase is therefore completed in $O(f(N)k)$ steps.

D. Transmission Phase

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 1. Each worm continues to move down its designated lane until it is level with the *entry point* of the first buffer on its right. It then takes a right hand turn and moves into this buffer through the entry point. These entry points are the circled processors shown in Figure 4.

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. Recall that 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 is even. As it stands, it takes such a worm $k + \sqrt{(f(N)+4)k} - 2$ 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 is odd. In that case, transmission takes $k + \sqrt{(f(N)+4)k}$ steps.

The algorithm is a deflection algorithm. Accordingly, during the time that this communication is taking place, we need to keep the storage snakes of each sub-mesh moving. We also need to make sure that the storage snakes do not block any worms that are being transmitted along the vertical/horizontal lanes. It is therefore necessary to add a delay to the extraction cycle using the method shown in Figure 3. Note that the addition of this delay will not affect the extraction phase in any way.

As we have just found out, it takes $k+\sqrt{(f(N)+4)k}$ steps to complete transmission of worms from even to odd sub-meshes and $k + \sqrt{(f(N)+4)k} - 2$ steps for transmissions from odd to even sub-meshes. The situation which occurs when a a worm is about to cross from one sub-mesh into another via the vertical lane is as shown in Figure 5 (the transmission via a horizontal lane is similar). We now explain why the situation described in Figure 5 is accurate. To understand the explanation, it is important to refer to Figures 1 and 4. We only consider transmission in the upwards direction as transmission in the downwards direction is symmetrical. In any sub-mesh, the worm which is transmitted in the upwards direction comes from the top right

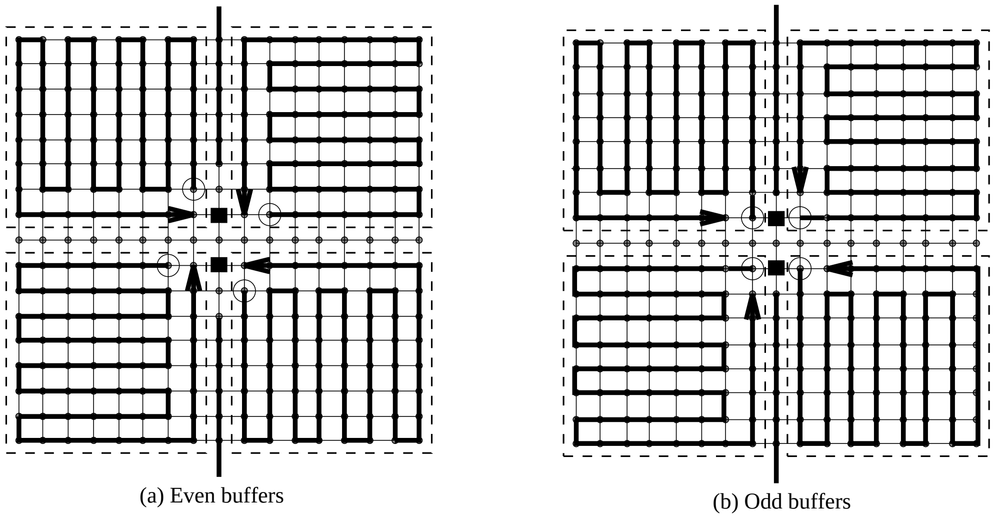


Fig. 4. The positions of the left and right storage snake tails when the top buffers are ready to transmit. The figure also shows the entry points (white circles) through which worms enter the buffers at the end of the transmission phase. Figure (a) shows the buffers of even supernodes. Figure (b) shows the buffers of odd supernodes.

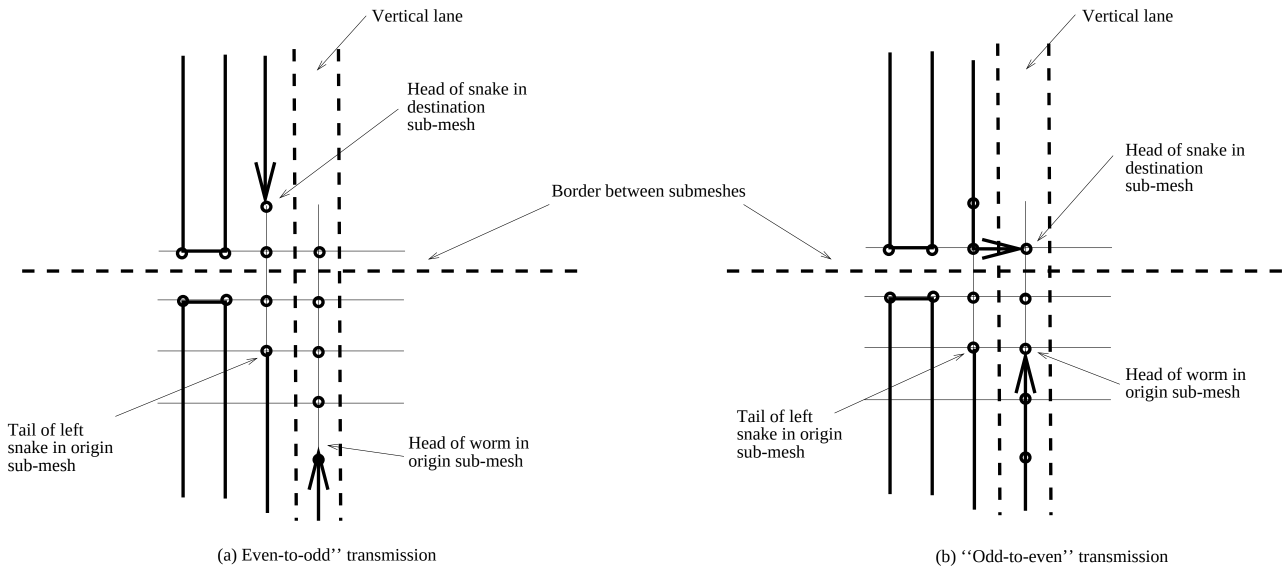

Vertical lane
Head of snake in destination sub-mesh
Border between submeshes
Tail of left snake in origin sub-mesh
Head of worm in origin sub-mesh
(a) Even-to-odd'' transmission
Vertical lane
Head of snake in destination sub-mesh
Head of worm in origin sub-mesh
Tail of left snake in origin sub-mesh
(b) ''Odd-to-even'' transmission

Fig. 5. 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. The horizontal dashed line indicates the border between one sub-mesh and the next. The vertical dashed lines are there to indicate the vertical lane. As can be seen, a delay must be added to the snake cycles in order to prevent a collision.

buffer and so it is this buffer that we now consider for the purpose of our explanation. As shown in Figure 4 the head of the top right buffered worm in an even sub-mesh is three processors behind the tail position of the left hand storage snake when transmission is ready to occur. The head of the worm which is being transmitted upwards therefore follows the tail of the left hand storage snake as it moves up the vertical lane, at a distance of three processors behind it. Finally, noting that the storage snake cycles of even sub-meshes are one step ahead of the cycles of odd sub-meshes, we conclude that the situation is as shown for the "even-to-odd" case. Similar reasoning may be applied to the "odd-to-even" case. This concludes the explanation.

In order to prevent the blocking of transmission and ensure that the snakes are in the correct position to receive incoming worms that are to be stored in the supernode's queue once transmission has occurred (the queueing phase), a delay of $k+4$ steps is added to the extraction cycle of all sub-meshes using the technique of Figure 3. Figure 6 shows the situation of Figure 5, after this delay have been added. Also, a two step delay is added during the buffering of all incoming worms in even sub-meshes. This delay will be justified in the subsection that describes the queueing phase. The important thing to state here is that the $k+4$ step delay which gives us the situation of Figure 6 ensures that the head of the left hand storage snake of the sub-mesh which was the destination of the transmission is exactly three processors away from the tail of the newly buffered worms when transmission is complete.

E. Queueing Phase

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

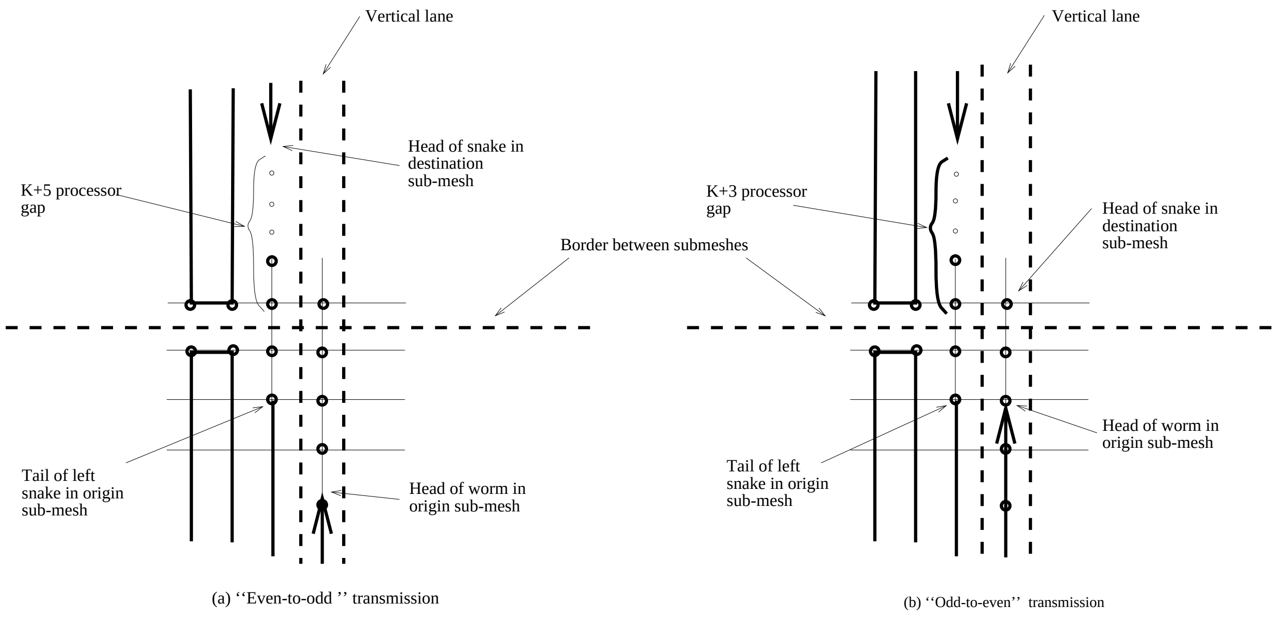

Vertical lane
Head of snake in destination sub-mesh
K+5 processor gap
Border between submeshes
Tail of left snake in origin sub-mesh
Head of worm in origin sub-mesh
(a) ''Even-to-odd '' transmission
Vertical lane
K+3 processor gap
Head of snake in destination sub-mesh
Head of worm in origin sub-mesh
Tail of left snake in origin sub-mesh
(b) ''Odd-to-even'' transmission

Fig. 6. 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. The horizontal dashed line indicates the border between one sub-mesh and the next. The vertical dashed lines are there to indicate the vertical lane.

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 always are 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 the worm does.The delays mentioned at the end of the last section ensure that the head of a buffered worm and the head of the snake into which it is to be stored are equidistant from the relevant command processor. In the case of even buffers this distance is 2 processors while, in the case of odd buffers this distance is 3 processors. This is exactly right considering that the storage snakes of even supernodes are one step ahead of the storage snakes of odd supernodes. All worms are inserted into storage once the snakes have completed their queueing cycle. The queueing phase is therefore completed in $O(f(N)k)$ steps.

F. 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 7. 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.

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 $(O(k))$ of extra steps using the method of Figure 3. We conclude that the absorption phase takes $O(f(N)k)$ steps to complete.

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

Theorem 1: 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 permutation deflection worm routing algorithm $\widehat{\mathcal{A}(n)}$ for M_n, which routes k-flit worms in $O((f(N)k)^{2.5}n)$ steps, where N satisfies $N = \frac{n}{\sqrt{(f(N)+4)k}+1}$ and $1 \leq k \leq \alpha \frac{n^2}{f(N)}$, for a small positive constant α.

Proof: The algorithm presented in the previous section requires $(\sqrt{(f(N)+4)k} + 1)^4$ 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 a fixed number of different phases. As we have seen, no phase requires more that $O(f(N)k)$ steps to complete. Each round is therefore completed in $O(f(N)kN)$ steps. The whole algorithm therefore takes $O((\sqrt{(f(N)+4)k}+1)^4 f(N)kN) = O((f(N)k)^{2.5}n)$ steps to complete.

An upper bound on the worm-size must be placed for the

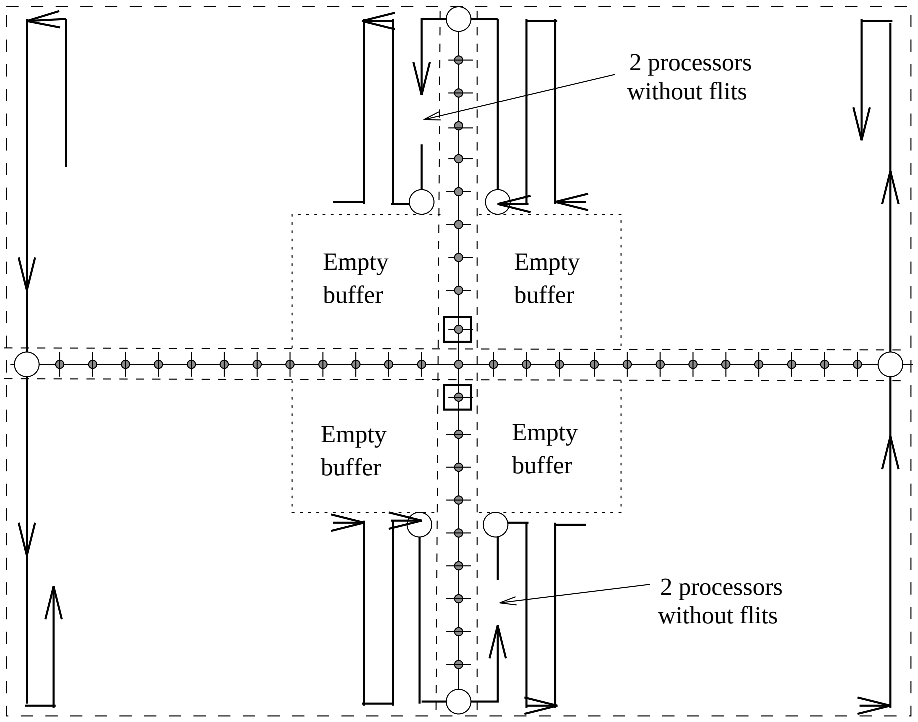


Fig. 7. The death cycle of the worms within a sub-mesh. The cycle passes completely through both storage areas. Any worm that has a target processor in the sub-mesh which does not lie on the cycle will be sent to its target by the appropriate white circled processor.

bound on the number of routing steps to be valid. It can be easily seen that $1 \leq k \leq \alpha \frac{n^2}{f(N)}$, for a small positive constant α, must hold. ■

V. Discussion

In this paper, we generalised the method of Newman and Schuster [10] for obtaining deflection worm routing algorithms on two-dimensional meshes by simulating known packet routing algorithms. We achieved this by relaxing the requirement that the simulated packet routing algorithm uses queues of at most 4 packet. Note that the proposed algorithm is not restricted only to permutations. 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. Since we have explained how to consume worms that arrive at their destination supernode, many-to-one routing problems can be easily solved. One-to-many routing problems can also be solved provided that we know how to store the worms that are to be routed into the storage snakes.

When we evaluate routing algorithms we usually focus on the resources used, that is, the number of routing steps and the queue size. One important aspect of the algorithms which is underestimated is their simplicity. The literature of routing algorithms is full of complex optimal algorithms (even with respect to the constants hidden in the big-Oh notation) and small queues. However, most of them are treated as impractical due to their complicated logic. It is obvious, that these algorithms cannot be used in simulations. On the other side, there are several near optimal algorithms that are extremely simple. For permutation routing, the algorithm of Han and Stanat [3] (in one of its versions) terminates after $5.5n$ routing steps ($1.5n$ steps move information between adjacent processors consisting only of one integer) and uses queues of at most 5 packets per node. For many-to-one routing, the algorithm of Makedon and Symvonis [9] requires $2n\sqrt{m} + o(n)$ routing steps and queues of at most 16 packets per node, where m is the maximum number of packets destined for a single node (the queue size can be further reduced but this will require a more complicated control structure). Both of the above algorithms appear to be inferior (within a constant factor) to existing algorithms. The algorithm of Han and Stanat [3] uses more routing steps that the optimum 2n-2 which was achieved in [8] and [11], and smaller queues. The algorithm of Makedon and Symvonis [9] uses larger queues (16 packets) compared to the algorithm of [14] which uses queues of 4 packets per node. However, both of the above mentioned algorithms are extremely simple and should be preferred in simulations.

We close by noting that the result presented in this paper as well as in the paper of Newman and Schuster [10] are unlikely to lead to efficient implementations of worm deflection routing algorithms. Rather, they demonstrate the existence of fast worm deflection algorithms. Further research must focus on practical algorithms. The indication from this paper and from [10] and [12] is that such algorithms exist.

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] P. Baran. On distributed communication networks. *IEEE Trans. on Commun. Systems*, CS-12:1–9, 1964.

[3] T. Han and D.F. Stanat. Move-and-smooth routing algorithms on mesh-connected computers. In *Proceedings of the 28th Allerton Conference*, pages 236–245, 1990.

[4] M. Kaufmann, H. Lauer, and H. Schroder. Fast deterministic hot-potato routing on processor arrays. In D.Z. Du and X.S. Zhang, editors, *Proceedings of the 5th International Symposium on Algorithms and Computation ISAAC '94 (Beijing, P.R. China, August 1994)*, LNCS 834, pages 333–341. Springer-Verlag, 1994.

[5] Kunde. Packet routing on grids of processors. *Algorithmica*, 9:32–46, 1993.

[6] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, San Mateo, CA 94403, 1991.

[7] F. T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing STOC '92 Victoria, British Columbia, Canada, May 4-6, 1992*, pages 77–96. ACM Press, 1992.

[8] F.T. Leighton, F. Makedon, and I.G. Tollis. A $2n-2$ step algorithm for routing in an $n \times n$ array with constant size queues. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures (Santa Fe, New Mexico, June 18-21, 1989)*, pages 328–335. ACM SIGACT, ACM SIGARCH, ACM Press, 1989.

[9] 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.

[10] Newman and Schuster. Hot potato worm routing via store-and-forward packet routing. *Journal of Parallel and Distributed Computing*, 30, 1995.

[11] S. Rajasekaran and R. Overholt. Constant queue routing on a mesh. *Journal of Parallel and Distributed Computing*, 15(2):160–166, June 1992.

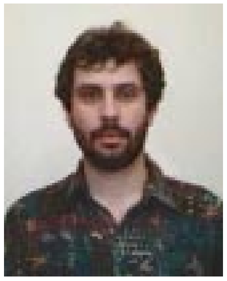
[12] A. Roberts and A. Symvonis. On deflection worm routing on meshes. In *Proceedings of the First IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA^{3}PP '95), Brisbane, Australia*, pages 375–378, apr 1995. (Also TR 490, October 1994, Basser Dept of Computer Science, University of Sydney.).

[13] 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.

[14] 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.

[15] 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. To appear in *IEEE Transactions on Computers.*

[16] M. Tompa. Lecture notes on message routing in parallel machines. Technical Report 94-06-05, Department of Computer Science and Engineering, University of Washington, June 1994.



Antonios Symvonis completed his undergraduate study in Computer Engineering and Information Sciences in 1987 at the University of Patras, Greece. He received his M.Sc. and Ph.D. degrees from the University of Texas at Dallas in 1989 and 1991, resp.
Since September 1991, he has been with the Basser Department of Computer Science at the University of Sydney. His principle research interests are packet routing algorithms, parallel processing, graph drawing algorithms, and graph theory.

Alan Roberts received the B.Sc. (Hons) degree majoring in Physics and Computer Science from the University of Sydney in 1990. He is currently working towards his Ph.D in Computer Science at the University of Sydney in the area of packet routing algorithms. He is also interested in object oriented software development.