Optimal algorithms for the many-to-one routing problem on two-dimensional meshes

Fillia Makedon* and Antonios Symvonis†

In this paper, we consider the many-to-one packet routing problem on the mesh parallel architecture. This problem has not been considered before. It models the communication pattern that occurs when many processors try to write on the same memory location on a concurrent-read concurrent-write shared memory parallel machine. We show that there is an instance of the many-to-one packet routing problem that requires $n\sqrt{k}/2$ routing steps to be solved, where k is the maximum number of packets a processor can receive. We give an algorithm that solves the problem in asymptotically optimal time. Furthermore, our algorithm uses queues of small constant size. This queue bound is very important since the ability to expand the mesh is preserved. Finally, we consider two variations of the many-to-one packet routing problem, namely, the case where k is not known in advance, and the case where combining the packets that are destined for the same processor is allowed.

many-to-one routing **mesh architectures** **packet routing algorithm**

An important task in the design of parallel computers is the development of efficient parallel data transfer algorithms. It is known as the packet routing problem, i.e. how to route messages (packets) from one processor to another. The routing algorithm of a parallel machine, usually called *router*, must be simple and fast. Another requirement is that the number of buffers, which are used in each processor to facilitate the routing, must be small and, if possible, independent of the size of the network. These buffers are required because queues can be created if two packets are competing for the same communication channel.

In this paper we study the many-to-one packet routing problem on the mesh architecture. We prove a lower bound for this problem and present an algorithm that is optimal in the worst case and uses small, constant size queues. We have chosen the mesh architecture because its simple and regular interconnection pattern makes it especially suited for VLSI implementations. An $n \times n$ mesh of processors is defined to be a graph $G = (V, E)$, where $V = \{(i, j) \mid 1 \leqslant i, j \leqslant n\}$ and an edge $e = ((i, j), (k, l))$ belongs to E if $|k - i| + |l - j| = 1$. The $n \times n$ mesh is illustrated in Figure 1. At any one step, each processor can communicate with all of its neighbours by the use of bidirectional links (channels). We define the distance between two processors $P_1 = (i, j)$ and $P_2 = (k, l)$ as distance $(P_1, P_2) = |k - i| + |l - j|$. If distance $(P_1, P_2) = 1$ then P_1 and P_2 are neighbours. The processors are assumed to work in a synchronous MIMD model.

Initially, in a many-to-one routing problem, each processor has exactly one packet that must be routed to another processor on the mesh. However, unlike the permutation packet routing problem, more than one packet might be destined for the same processor. A typical problem that can be modelled as a many-to-one routing problem is the process of writing on the same memory location on a CRCW (concurrent-read concurrent-write) parallel machine. In this problem, each processor holds a portion of the shared memory. If more than one processor wants to write on the same memory location L, then these processors will all route their data to the processor that holds memory location L. We can easily see that the above situation on a CRCW parallel machine corresponds to the many-to-one routing problem.

To the best of our knowledge, no previous work exists

*Department of Mathematics and Computer Science, Dartmouth College, 6188 Bradley Hall, Room 305, Hanover, NH 03755-3551, USA. E-mail: makedon@dartmouth.edu
†Basser Department of Computer Science, University of Sydney, Sydney, NSW 2006, Australia. E-mail: symvonis@cs.su.oz.au
Paper received: 12 June 1992. Revised: 6 November 1992

0141-9331/93/060361-07 © 1993 Butterworth-Heinemann Ltd

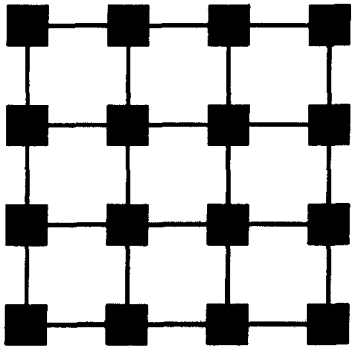


Figure 1. The two-dimensional mesh

on the many-to-one routing problem for the mesh connected array parallel architecture. On the other hand, the permutation routing problem has been studied. An optimal algorithm for the 1-1 packet routing problem on the mesh that takes $2n - 2$ steps and uses queues of 1008 was derived by Leighton *et al.*[1]. Later, the queue size was reduced to 112 by Rajasekaran and Overholt[2]. The permutation routing problem can be used to model communication patterns that occur in EREW (exclusive-read exclusive-write) parallel machines. A modification of the permutation routing problem, called flit-serial routing, considers the case where the packets are broken into a string of subpackets, called 'flits'. The problem here is to route the 'snake' and flits quickly and with small queues. Also, the sequence of the flits that constitute a packet cannot be broken during the routing. The authors have studied this problem for the mesh and the torus[3] (a mesh with wrap-around connections).

The remainder of the paper is divided into sections as follows: in the next section, we show a lower bound on the number of communication steps required to solve the many-to-one routing problem for the mesh-connected array parallel architecture. In the following section, we present an asymptotically optimal algorithm for the many-to-one routing problem that uses queues of size at most $2n$. We then bound the queue size to 16 packets per processor. Next, we consider two variations of the many-to-one routing problem. We conclude with open questions.

THE LOWER BOUND

Theorem 1 Given an $n \times n$ mesh of processors, where initially each processor holds exactly one packet, there is a many-to-one routing problem that requires $\Omega(n\sqrt{k})$ routing steps to be solved, where k is the maximum number of packets a processor can receive.

Proof To prove the lower bound, let us consider the following situation. Assume that there exists a set of processors such that all the processors in that set receive exactly k packets. Since the total number of packets in the mesh is n^2, there are exactly n^2/k such processors. Furthermore, assume that these processors are located at the south-east corner of the mesh and, more specifically, inside an $(n/\sqrt{k}) \times (n/\sqrt{k})$ south-east square submesh M_1 (Figure 2). Note that submesh M_1 contains exactly n^2/k processors. Assume that all packets in the network want to enter into this submesh. There are a total of $n^2 - (n^2/k)$ such packets and these packets can enter into submesh

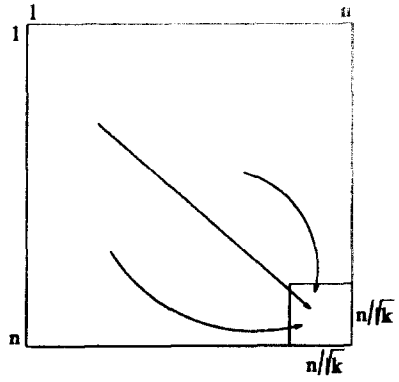


Figure 2. All packets are destined for the south-east submesh

M_1 only through $2n/\sqrt{k}$ channels. Thus, any solution to this routing problem requires at least:

$$\frac{n^2 - n^2/k}{2n/\sqrt{k}} = \frac{n^2\sqrt{k}(k-1)}{2nk} = \frac{n\sqrt{k}}{2} - \frac{n}{2\sqrt{k}}$$

routing steps in order to route all packets to their destinations. This indicates that we need at least $\Omega(n/\sqrt{k})$ routing steps.

AN OPTIMAL ALGORITHM

In this section we present an asymptotically optimal algorithm that solves the many-to-one problem such that each processor uses queues of size at most n packets. In the next section we will modify the algorithm so that the queue size is bounded by a constant. This is very important since the algorithm can then be used for networks of any size. If the queue size were a function of n, then any attempt to expand the network would require extra buffers to be added to all the processors of the existing network. This is something to be avoided.

The algorithm

A high level description of the routing algorithm is given. The algorithm consists of one colouring phase (step 1) and two routing phases (steps 2 and 3). The colouring phase will be refined later.

Algorithm 'route'

1. Out of those packets that are destined for the same row, colour (using algorithm *Colour* below) exactly $n/\sqrt{k}$ of them 'black' (if they exist) and the rest 'white'.
2. Route the black packets as follows:
   (a) Route the black packets vertically (along the columns) to the correct row.
   (b) Route the black packets horizontally (along the rows) to their destination.
3. Route the white packets as follows:
   (a) Route the white packets horizontally (along the rows) to the correct column.
   (b) Route the white packets vertically (along the columns) to their destination.

We say that a black (white) packet that is moving along a column (row) during step 2 is executing its 'phase I

routing'. Similarly, we say that a black (white) packet that is moving along a row (column) during step 3 is executing its 'phase II routing'.

It is clear that the algorithm terminates and all packets will reach their destination. The way steps 2 and 3 are performed is also clear. At each time instance, each processor advances one of the packets it has, out of those that want to travel in a given direction, towards that direction. If it has a packet to send, it always does so.

Step 1 needs further refinement. We have to specify the way we colour the packets. Furthermore, we have to colour them quickly so that the colouring phase adds to the total routing time very small terms which can be ignored from the overall time complexity. We use algorithm *Colour* to do the colouring. This algorithm uses a sorting algorithm as its first step, which sorts all the packets of the mesh using a special indexing scheme called row_major-snake_like-column_order indexing. Before we describe algorithm *Colour*, however, we need to explain how the packets are arranged after the sorting if the row_major-snake_like-column_order indexing scheme is used.

Suppose that we start with a square mesh of processors, where each processor holds a packet that is destined for another processor on the mesh. Each packet has a destination address that is a pair (row, column) of integers. If the packets are sorted according to their destination in a row_major-snake_like-column_order, then the 'smallest' packets will end up at processor (1, 1) while the rest form a snake-like sequence positioned along the columns. The term 'row_major' indicates that all packets that are destined for the same row form a sequence in the 'snake' denoted by S_i. Similarly we define the column_major-snake_like-row_order that we will use later in the paper. Figure 3 shows a 5 × 5 mesh before and after the sorting of its packets in row_major-snake_like-column_order. The figure also illustrates the notion of a sequence.

Algorithm 'colour' (step 1 of algorithm 'route')

1. Sort the mesh in row_major-snake_like-column order.
2. In each sequence S_i, where $1 \leqslant i \leqslant n$, locate the 'first' packet, called the leader. Let this packet be p_i.
3. Colour all packets white.
4. For all sequences S_i, $1 \leqslant i \leqslant n$, in parallel do: Starting from p_i, colour $n\sqrt{k}$ packets (if they exist) of sequence S_i black.

Step 1 is the most critical step in algorithm *Colour*. After the sorting has finished, the packets are partitioned into n consecutive sequences (some of which might be empty) such that packets that are destined for the same row belong to the same sequence. After the sorting and the identification of the 'leader' in each sequence have taken place, 'special' packets are allowed to travel along each sequence and colour the first $n\sqrt{k}$ ordinary packets black (step 4). We will see later how step 4 takes place.

(2,5)	(4,1)	(1,1)	(2,3)	(1,1)
(2,5)	(1,1)	(2,3)	(1,1)	(2,5)
(1,2)	(2,5)	(2,5)	(1,1)	(1,1)
(2,5)	(1,2)	(4,1)	(4,1)	(1,2)
(1,1)	(4,1)	(1,1)	(1,2)	(1,1)

(1,1)	(1,2)	(1,2)	(2,5)	(2,5)
(1,1)	(1,1)	(1,2)	(2,5)	(4,1)
(1,1)	(1,1)	(1,2)	(2,5)	(4,1)
(1,1)	(1,1)	(2,3)	(2,5)	(4,1)
(1,1)	(1,1)	(2,3)	(2,5)	(4,1)

Figure 3. A 5 × 5 array before and after sorting in row_major-snake_like-column_order. The three sequences destined for rows, 1, 2 and 4 are shown

Time analysis

We first analyse algorithm *Colour*. The sorting step can be performed in $3n + o(n)$ routing steps using the sorting algorithm of Schnorr and Shamir[4]. Let us now consider the time it takes to locate the leader in each sequence. Lemma 1 shows that finding the leader of a sequence takes one routing step.

Lemma 1 In an $n \times n$ array that is sorted in row_major-snake_like-column_order, a processor can find out if it is holding the leader of a sequence in one routing step.

Proof The following strategy proves the lemma. Each processor sends a copy of its packet to the processor immediately after it in the 'snake' ordering. After this, each processor compares the packet it received with its own packet. If both packets are destined for the same row, then the processor is not holding the leader of the sequence to which it belongs. If the packets are destined for different rows, then this processor is holding the leader of the sequence. The processor at location (1, 1) of the mesh always holds a leader. From the above discussion, it is obvious that a processor can find out if it holds a leader in one routing step.

Next, we give a lemma that bounds the time it takes to colour the first $n\sqrt{k}$ packets of each sequence black (step 4 of algorithm *Colour*). Let M be an $n \times n$ mesh such that each processor holds a packet and the packets are sorted in row_major-snake_like-column_order.

Lemma 2 Given a mesh M and k, the maximum number of packets a processor can receive, we can colour black the first $n\sqrt{k}$ packets of each sequence S_i in $n + \sqrt{k}$ steps, $1 \leqslant i \leqslant n$.

Proof To prove Lemma 2, we must describe how to colour the first $n\sqrt{k}$ packets (if they exist) of any sequence in $n + \sqrt{k}$ steps. Let P_i^0 be the processor that holds the leader of sequence S_i. We designate a special packet of type 'searcher' to start moving from P_i^0 to the right along the row. Let P_i^1 be the processor it meets at the next position on the row. If P_i^1 is also holding a packet destined for row i, then all packets of the sequence that exist on the processors between P_i^0 and P_i^1 are destined for the same row and must be coloured black. We now let two special packets of type 'painter' start, one from P_i^0 along the 'snake' toward P_i^1 and the other from P_i^1 along the 'snake' toward P_i^0. During their trip, these two painters colour all the packets on their route black. After this, the searcher moves to the right again and the same procedure is repeated. Since we want to colour up to $n\sqrt{k}$ packets, we allow the searcher of each sequence to move up to $\sqrt{k}$ positions to the right. So, in the worst case, the movement of the searcher takes $\sqrt{k}$ steps. The movements of the painters that we initiated last will finish after at most n steps. Thus the colouring of the packets can be done in $n + \sqrt{k}$ steps.

Note that we can omit using the second painter. The result will be that the colouring finishes in at most $2n + \sqrt{k}$ steps. This does not affect the final complexity of the algorithm by more that a constant factor.

From Lemmas 1 and 2, we conclude that algorithm *Colour* takes $4n + \sqrt{k} + o(n)$ routing steps.

Now we analyse the routing phases (steps 2 and 3 of algorithm *Route*). We observe that steps 2(a), 2(b), 3(a) and 3(b) can be reformulated as special cases of the following problem: given m packets that are distributed along a chain of n processors and such that each one has a destination in the chain associated with it, route the packets as quickly as possible. This problem, as well as its variants, is very well studied in several papers[3, 5, 6]. An optimal algorithm exists that completes the routing after at most $m + n$ steps. The worst case situation occurs when all packets are initially located at the processor at one end of the chain and are destined for the same processor at the other end of the chain.

Let us now consider steps 2(a) and 3(a) of algorithm *Route*. Since there are at most $d \leqslant n$ packets in each column and row respectively, at most n routing steps will be needed in the worst case. (We might need more than d because of the distance limit. A packet from one end of the chain (row or column of the mesh) might be destined for the processor at the other end.)

In step 2(b) we have at most $n\sqrt{k}$ packets in each row. This is because algorithm *Route* forces that many packets to be in each row (if they exist) by colouring them black. Let w be the maximum number of white packets in any column at the beginning of step 3(b). Then the number of routing steps needed for step 3 of algorithm *Route* is $\max(n\sqrt{k}, w) + n - \sqrt{k}$. In the following lemma we prove that $w < n\sqrt{k}$.

Lemma 3 The number of white packets in every column of the mesh at the beginning of step 3(b) of algorithm *Route* is less than $n\sqrt{k}$.

Proof Recall that algorithm *Route* initially colours all packets white. Then it colours up to $n\sqrt{k}$ packets of each sequence S_i black. All the remaining packets of each sequence are still coloured white. But how many sequences still have white packets? Since we started with exactly n^2 packets in the mesh, there must be less than $n^2/(n\sqrt{k}) = n/\sqrt{k}$ such sequences. Assume that there exists a column such that k packets from each of these sequences with white packets are destined for this column. Since there are less than $n/\sqrt{k}$ such sequences, the total number of white packets in any column is less than $(n/\sqrt{k})k = n\sqrt{k}$.

From Lemma 3, we can conclude that each of steps 2 and 3 of algorithm *Route* takes at most $n(\sqrt{k} + 1) - \sqrt{k}$ routing steps. Thus, the total number of steps required by algorithm *Route* is:

$$4n + \sqrt{k} + o(n) + 2(n + n(\sqrt{k} + 1) - \sqrt{k}) = O(n\sqrt{k})$$

routing steps. From Theorem 1, we conclude that the algorithm is optimal.

Let us consider the queue size. The queue size required at each processor of the mesh is of size at most n packets per direction (horizontal or vertical). This is because, at the end of step 2 of the algorithm, we might have at some processor at position (i, j) of the mesh all packets that were initially in column j. There are at most n such packets. The same holds for the white packets that were routed through the rows (actually, at most $n - \sqrt{k}$ packets are queued in any processor). Thus we have the following theorem.

Theorem 2 Given an $n \times n$ mesh of processors, where each processor has exactly one packet destined for another processor of the mesh, and k, the maximum number of packets a processor can receive, the many-to-one routing problem can be solved in $\Theta(n\sqrt{k})$ routing steps with the use of queues of size at most n packets.

BOUNDING THE QUEUE SIZE

In this section, we describe how to modify algorithm *Route* so that the size of the queues used will be bounded by a constant. If we can bound the queue size by a constant, then we are able to apply algorithm *Route* on networks of any size.

Structure of each processor

Before we proceed to describe how to modify algorithm *Route* in order to achieve constant size queues, we present the structure that a processor must have in order to be able to execute the modified algorithm. Of course, since we are interested in the queue size, we show only the structure of the buffer area of each processor.

Our routing algorithm will first route the black and then the white packets. So the buffers that are used in the routing of the black packets can be reused. Figure 4 shows a processor with the buffers that are used in the routing of black packets. (Remember that black packets are routed vertically to the correct row, and then horizontally to their destination.)

With each of the north and the south channels we associate two buffers of each of the following three types: TURN_L, TURN_R and STRAIGHT buffer types. Buffers of type TURN_L accommodate only packets that want to enter through the specific channel the buffers are associated with, and want then to turn to the left. Similarly, buffers of type TURN_R accommodate only packets that want to enter through the specific channel

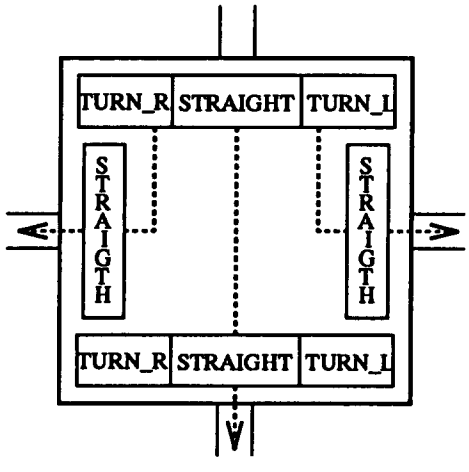


Figure 4. The buffers used in the routing of black packets. Dashed lines indicate the path that a packet in a particular type of buffer will follow

the buffers are associated with, and want then to turn to the right. Observe that at the time a packet enters a buffer of type TURN__L or TURN__R, it starts to execute its phase II routing. Buffers of type STRAIGHT hold packets that want to exit the processor through the opposite channel. With each of the east and the west channels, we associate two buffers of type STRAIGHT. In total, each processor has 16 buffers, each capable of storing one packet. It is obvious that these buffers can be reused for the routing of the white packets. The corresponding figure can be obtained by rotating Figure 4 by 90°. In the following section, it will be clear why we need two buffers of each buffer type.

Queueing strategy

In order to maintain constant queues we use the following strategy: a packet will be transmitted to the next processor on its route only if there is enough space to accommodate the packet at the next processor. Depending on the fact that the given packet might want to turn at the next processor, it will be placed on a buffer of the appropriate type. So, a buffer of a particular type must be free when the packet arrives.

Let us now explain why we use two buffers of each buffer type. Without loss of generality, concentrate on a particular channel that connects processors A and B, and the queues of a particular type created at processor B because of packets it receives from processor A. Our general queueing strategy will be the following. Processor B examines the information it received at step $t - 1$ from its neighbours and decides what packets it will transmit during step t. It then removes these packets from its queues. Then it checks its queues and, if there is space available, sends a signal to processor A requesting a packet. The natural questions now are, how large a queue needs to be and how much space in the queue of B must be available in order for it to send a signal. To clarify why we need two buffers, assume that processor A always has a packet to transmit. Also observe that, if processor B sends a signal at step t, then it will receive the packet at the end of step $t + 1$, and so it will be able to transmit it at step $t + 2$. If we want to have processor B always transmitting, then it must have one packet to transmit during step $t + 1$. So, processor B must have had one packet in its queue (if possible) when it sent the signal to processor A at step t. Thus, a signal will be sent if there is at most one packet in the queue.

So far, we have answered the second question. Now that we have established that at least queues of size one are needed, let us justify why we must have space for an extra packet. We considered before the case where, at step t, processor B sends a signal to processor A and it also transmits a packet from its queue at step $t + 1$. What if it cannot transmit at step $t + 1$ (because it did not receive a signal from its neighbours)? Obviously, at the end of step $t + 1$, two packets will be at processor B (if processor A responds): the packet that already existed and the packet just received from processor A. From the above discussion it is clear that two packets of each buffer type are needed for every channel.

Resolving conflicts

The queueing strategy described in the previous section is not enough to guarantee constant size queues. During the routing we may face the following situation. A processor receives a signal from one of its neighbours that allows it to transmit a packet. What if there is more than one packet waiting to be transmitted to that neighbour? Obviously, we have to assign priorities. There are two kinds of conflict that we have to resolve:

1. Conflicts between packets that are in buffers of type TURN__L (or TURN__R) and of type STRAIGHT. Both packets, at the next processor, will use buffers of type STRAIGHT. We give priority to the packets that are in buffers of type TURN__L (or TURN__R). If there are packets in buffers of both of these types, the decision is taken arbitrarily. The reason we give priority to packets that are in buffers of type TURN__L (or TURN__R) is that we want to make the packets that are on the front or the end columns of a sequence (relative to the destination processor) move faster.
2. Conflicts between packets that are executing phases I or II and want to move straight. We use a FIFO policy here. The packet that enters first will exit first. This is done in order to forbid 'overpassing' between packets. One reason for the sorting of packets is that we want them to flow in a regular manner. If 'overpassing' is allowed, we will have problems in analysing the time required for the algorithm.

Routing algorithm

We now give a very high-level description of the algorithm that solves the many-to-one routing problem and uses constant queues.

Algorithm 'route__with__constant__queues'

1. Colour the packets black and white as in algorithm *Route*.
2. For each processor, repeat until the routing of black packets is completed:
   (a) From the information it received in the previous routing step and, using the methods to resolve conflicts described in the previous section, the processor decides which packets to transmit.
   (b) Using the criteria in the previous section, the processor determines which of its queues can receive packets at the next routing step. During the next routing step, the processor will send signals to its neighbours that allow or forbid them to transmit packets destined for its queues.
   (c) The processor transmits the packets from step 2(a) concatenated with the signals it decided to send at step 2(b).
3. Sort the white packets in column__major-snake__like-row__order.
4. Route the white packets as in step 2.

While it is clear why we need steps 1, 2 and 4, some justification is required for the use of step 3. We need it in order to make the packets flow regularly. If we do not perform it, we might face the situation where packets that are waiting to turn left or right delay packets behind them that want to go straight on. Moreover, when this happens, the destinations of the packets that want to go straight on will not follow any pattern. This will complicate our analysis. After sorting the white packets, the analysis of their routing is identical to that of the black packets.

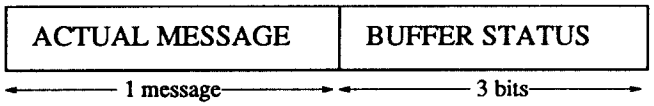


Figure 5. The structure of a packet

Observe that the information concerning the buffers that must be transmitted consists of only three bits. Figure 5 shows the structure of a packet transmitted during step 2(c). If a particular bit of the signal segment is set to '1', then this is interpreted as an invitation for packets.

We prove the following theorem.

Theorem 3 Given an $n \times n$ mesh of processors, where each processor has exactly one packet destined for another processor of the mesh, and k, the maximum number of packets a processor can receive, the many-to-one routing problem can be solved in $\Theta(n\sqrt{k})$ routing steps with the use of a buffer area of 16 packets per processor.

Proof Since nothing changes during the colouring strategy, we still have that at most $n\sqrt{k}$ packets will be routed during their phase II routing in any row or column. Without loss of generality, we concentrate on black packets of some sequence S_i. Generally, this sequence forms a 'snake' that occupies several columns. If the first and the last column are not completely occupied by sequence S_i, we ignore them for the moment and we say that these columns are not 'complete'. Consider the rest of the packets. All of them are destined for row i. Since their movement does not interfere with the movement of white packets, the routing can be completed in at most $n\sqrt{k} + n$ steps.

Consider now the packets that occupy the first and the last column of the sequence S_i which we ignored before. These packets might be waiting behind packets of other sequences, as a consequence of our FIFO policy, and thus they might reach their row destination later than the packets in 'complete' columns. However, they will all reach their row destination after $n\sqrt{k} + 2n$ steps. To see why we need so many steps, observe that after $n\sqrt{k} + n$ steps the packets that were in front of them are all in their row destination. Now, the packets in the first and last column of sequence S_i can move towards their row destination. They need at most n steps to reach their row. From this point on, n more steps are enough for these packets to reach their destination.

By summing the number of routing steps required for routing, sorting and colouring, we conclude that algorithm *Route_with_Constant_Queues* will complete the routing of the black packets in $O(n\sqrt{k})$ routing steps. The same amount of time is needed for the routing of the white packets. From the discussion in the previous section, we know that a buffer area of 16 packets per processor is used. This completes the proof.

VARIATIONS OF THE PROBLEM

In this section we study two variations of the many-to-one packet routing problem. In the first variation we assume that k, the maximum number of packets a processor can receive, is not known ahead of time. In the second variation, we consider the many-to-one routing problem when packets going to the same processor are allowed to combine before they reach that processor.

When k is not known in advance

In the many-to-one routing problem we have considered so far, the maximum number of packets k a processor can receive is known in advance. What if it is not? The obvious solution is to compute k. This can be done by first computing the maximum length out of all of the sequences which are created after sorting. We already know, from Lemma 1, that the leader of each sequence can be identified in only one routing step. Furthermore, with a procedure similar to that described in Lemma 2, we can count the number of packets in each sequence and store this number in the processor holding the last packet of the sequence. This task takes at most $2n$ steps.

We then perform a *Maximum* operation over all such values of the $n \times n$ array. This can be done as follows. Route all computed maximum values horizontally to the middle column of the mesh. The processors there select the maximum of the incoming values. This takes exactly $n/2$ steps. Then a *Maximum* operation is performed again among the values of the middle column. This also takes $n/2$ steps and the maximum value is stored at the processor located at the centre of the mesh. We now use the reverse procedure to broadcast the computed maximum value to all processors of the mesh, and thus to the leader of each sequence. Following this, algorithm *Route* proceeds with its colouring phase. So, in order to compute k, we used $4n$ extra routing steps. By summing all the above terms we find that after $O(n\sqrt{k})$ steps the routing is completed. Thus, we can conclude with Theorem 4.

Theorem 4 Given an $n \times n$ mesh of processors where each processor has a packet destined for another processor of the mesh, the many-to-one routing problem can be solved in $\Theta(n\sqrt{k})$ routing steps with a buffer area of 16 packets per processor, where k is the maximum number of packets a processor can receive and is determined during the course of the routing algorithm.

Many-to-one packet routing with combining

In this section we consider again the many-to-one packet routing problem. However, instead of routing all the packets independently, we now study a way to combine the packets destined for the same processor and then route the remaining packets. We define a subsequence s_{ij} to be a list of all packets that are destined for the processor at location (i, j). In total, there are n^2 subsequences (some of which might be empty). Observe that each sequence that was created by the sorting phase of the algorithm consists of at most n consecutive subsequences. We can solve the routing problem by simply routing the leaders of each subsequence of the sorted array. We observe that we may have up to n^2 such leaders. With this formulation, our problem is no longer a many-to-one packet routing problem: it is a permutation problem. We can still apply our algorithm to solve the problem. However, we can skip the colouring phase since we have already performed the sorting, and, since $k = 1$, all packets are coloured black. An alternative solution is to

use the algorithm of Rajasekaran and Overholt[2] to route the leaders in $2n$ steps with constant queues. In any case, we need $5n + o(n)$ steps, $3n + o(n)$ steps for sorting and $2n$ steps for routing. Note that this method of combining is very important in the case where 'write-conflicts' in a CRCW parallel machine are resolved in an arbitrary manner, i.e. among all processors that want to write in a common memory location, one processor is arbitrary chosen to do so. This result is reflected in Theorem 5.

Theorem 5 Any routing problem on an $n \times n$ mesh of processors where initially each processor has at most one packet destined for another processor on the mesh can be solved within $5n + o(n)$ routing steps and constant queues, provided that combining of packets destined for the same processor is allowed.

CONCLUSION

In this paper, we have presented an asymptotically optimal algorithm for the many-to-one packet routing problem on a mesh connected parallel computer that uses queues of 16 packets per processor. The importance of the problem comes from its ability to model the communication patterns that occur in a CRCW parallel machine. An interesting open problem is to derive an algorithm that matches exactly the lower bound obtained in the second section (our algorithm is optimal within a factor of 4 and can be modified to be within a factor of 2 if the routing of the black and white packets is done simultaneously) or to improve the lower bound to $n\sqrt{k}$ routing steps.

REFERENCES

1 **Leighton, F T, Makedon, F and Tollis, I G** '$A\ 2n$-2 algorithm for routing in an $n \times n$ array with constant size queues' *Proceedings of ACM Symposium on Parallel Algorithms and Architectures, SPAA '89* (June 1989) pp 328–335

2 **Rajasekaran, S and Overholt, R** 'Constant queue routing on a mesh' *J. Parall. Distrib. Comput.* Vol 15 (1992) pp 160–164

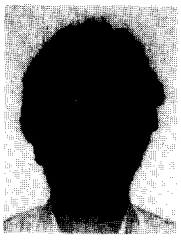
3 **Makedon, F and Simvonis, A** 'On bit-serial packet routing for the mesh and the torus' *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation* IEEE Computer Society Press (1990) pp 294–302

4 **Schnorr, C P and Shamir, A** 'An optimal sorting algorithm for mesh connected computers' in *Proceedings 18th ACM Symposium on Theory of Computing* (1986) pp 255–263

5 **Krizanc, D, Rajasekaran, S and Tsantilas, Th** 'Optimal routing algorithms for mesh-connected processor arrays' in **Reif, J (Ed)** *VLSI Algorithms and Architectures (AWOC '88)* Lecture Notes in Computer Science 319 Springer-Verlag, Berlin (1988) pp 411–422

6 **Kunde, M** 'Routing and sorting on mesh-connected arrays' in **Reif, J (Ed)** *VLSI Algorithms and Architectures (AWOC '88)* Lecture Notes in Computer Science 319 Springer-Verlag, Berlin (1988) pp 423–433

Fillia Makedon received her PhD in computer science at Northwestern University in Evanston, Illinois in 1982. Following this, she served on the faculty of the Illinois Institute of Technology in Chicago and at the University of Texas at Dallas. She is currently Associate Professor in the Department of Mathematics and Computer Science at Dartmouth College. Her main research interests are in algorithm optimization, CAD heuristics, parallel computation and algorithm visualization.



Antonios Symvonis finished his undergraduate study in computer engineering and information sciences in 1987 at the University of Patras, Greece. He received MSc and PhD degrees from the University of Texas at Dallas in 1989 and 1991. His PhD dissertation was on packet routing algorithms. Since September 1991, he has been a Lecturer at the Basser Department of Computer Science at the University of Sydney. His principal research interests are packet routing algorithms, interconnection networks, parallel processing and graph theory.