Journal of Graph Algorithms and Applications
http://www.cs.brown.edu/publications/jgaa/
vol. 5, no. 5, pp. 17–38 (2001)

New Bounds for Oblivious Mesh Routing*

Kazuo Iwama †
School of Informatics
Kyoto University
Kyoto 606-8501, JAPAN
http://www.lab2.kuis.kyoto-u.ac.jp/~iwama/index.html
iwama@kuis.kyoto-u.ac.jp

Yahiko Kambayashi ‡
School of Informatics
Kyoto University
Kyoto 606-8501, JAPAN
http://www.isse.kuis.kyoto-u.ac.jp/usr/yahiko/yahiko-e.html
yahiko@i.kyoto-u.ac.jp

Eiji Miyano §
Kyushu Institute of Design
Fukuoka 815-8540, JAPAN
http://www.kyushu-id.ac.jp/~miyano/index.html
miyano@kyushu-id.ac.jp

Abstract

We give two, new upper bounds for oblivious permutation routing on the mesh networks: Let N be the total number of processors in each mesh. One is an $O(N^{0.75})$ algorithm on the two-dimensional, $\sqrt{N} \times \sqrt{N}$ mesh with constant queue-size. This is the first algorithm which improves substantially the trivial $O(N)$ bound for oblivious routing in the mesh networks with constant queue-size. The other is a $1.16\sqrt{N} + o(\sqrt{N})$ algorithm on the three-dimensional, $N^{1/3} \times N^{1/3} \times N^{1/3}$ mesh with unlimited queue-size. This algorithm allows at most three bends in the path of each packet. If the number of bends is restricted to minimal, i.e., at most two, then the bound jumps to $\Omega(N^{2/3})$ as was shown in ESA'97.

Communicated by T. Nishizeki, R. Tamassia and D. Wagner:
submitted January 1999; revised April 2000 and October 2000.

*A preliminary version of this paper was presented at the 6th European Symposium on Algorithms (ESA'98).

†Supported in part by Scientific Research Grant, Ministry of Japan, 09480055 and 08244105, and Kayamori Foundation of Information Science Advancement, Japan

‡Supported in part by Scientific Research Grant, Ministry of Japan, 08244102

§Supported in part by Scientific Research Grant, Ministry of Japan, 10780198 and 12780234, and The Telecommunications Advancement Foundation, Japan.

1 Introduction

An algorithm for *packet routing* has to determine each packet's path through a network by using various information, such as source addresses, destinations, and the configuration of the network. So far a great deal of effort has been devoted to the design of efficient routing algorithms and there is a large amount of literature even if we focus our attention on deterministic, *permutation* routing, in which the destinations of packets are all different. The efficiency of a routing algorithm is generally measured by its *running time* and its *queue-size* of each processor, where the former is the total number of communication time-units the algorithm requires to route all packets to their destinations, and the latter is the maximum number of packets the processor temporally can hold at the same time during routing.

A very popular strategy for permutation routing is *oblivious* routing, in which the path of each packet is completely determined by its initial and final positions and is not affected by other packets it encounters. Hence, it is hard to avoid path-congestion in the worst case and it often takes much more time than it looks. Several lower-bounds which are rather surprising are known on the running time of oblivious permutation routing on standard *mesh* networks, each of which has N processors connected via point-to-point connections: For example: (i) An $\Omega(N)$ lower bound is known for oblivious permutation routing on any k-dimensional, constant queue-size mesh network including N processors, where k may be any constant [8]. (Note that an $O(N)$ upper bound can be achieved for permutation routing even on one-dimensional meshes (linear arrays) including N processors, i.e., increasing dimensions in meshes does not work in the worst case.) (ii) An $\Omega(N^{2/3})$ lower bound is known for oblivious permutation routing on *three-dimensional*, unbounded queue-size meshes including $N^{1/3} \times N^{1/3} \times N^{1/3}$ processors [5], which is much *worse* than the $O(\sqrt{N})$ bound for oblivious permutation routing on two-dimensional, unbounded queue-size meshes including $\sqrt{N} \times \sqrt{N}$ processors. (iii) An $\Omega(\sqrt{N})$ lower bound for oblivious permutation routing on any constant-degree, unbounded queue-size, N processor network [2, 3, 6]. It should be noted, however, that these lower bound proofs needed some supplementary conditions that might not seem so serious but are important for the proofs. In this paper, it is shown that the above lower bounds do not hold any more if those supplementary conditions are slightly relaxed.

More precisely, Krizanc needed the *pure condition* other than the oblivious condition to prove the $\Omega(N)$ lower bound in [8]. Roughly speaking, the pure condition requires that each packet must move if its next position is empty. Krizanc gave an open question, i.e., whether his linear bound can be improved by removing the pure condition. In this paper we give a positive answer to this question: It is shown that there is an $O(N^{0.75})$ oblivious algorithm for permutation on 2D, $\sqrt{N} \times \sqrt{N}$ meshes with constant queue-size, and there is an $O(N^{5/6})$ oblivious algorithm on 3D, $N^{1/3} \times N^{1/3} \times N^{1/3}$ meshes with constant queue-size. The oblivious condition used in [8] is a little more stronger than the normal one, called the *source-oblivious condition*. That is also satisfied

by our new algorithm, i.e., we remove only the pure condition in this paper. Note that this $\Omega(N)$ lower bound is quite tough; it still holds even without the oblivious condition; the *destination-exchangeable* strategy also implies the same lower bound if the queue-size is bounded above by some constant [4]. Our new bound can be extended to the case of general queue-size k, namely, it is shown that there is an $O(N^{0.75}/\sqrt{k})$ algorithm for 2D, $\sqrt{N} \times \sqrt{N}$ meshes of queue-size k, and there is an $O(N^{5/6}/\sqrt{k})$ algorithm for 3D, $N^{1/3} \times N^{1/3} \times N^{1/3}$ meshes of queue-size k, while an $\Omega(N/k(8k)^{5k})$ lower bound was previously known for any constant degree, k-queue-size network including N processors under the pure condition [8]. For 2D meshes, if we set $k = \sqrt{N}$, then that is equivalent to unbounded queue-size. Our bound for this specific value of k is $O(N^{0.75}/N^{0.25}) = O(\sqrt{N})$, which matches the lower bound of [2, 3, 6].

Our second result concerns with 3D meshes: In [5] an important exception was proved against the well-known superiority of the 3D meshes over the 2D ones; oblivious permutation routing requires $\Omega(N^{2/3})$ steps over the 3D meshes including N processors under the following (not unusual, see the next paragraph) condition: The path must be shortest and be as straight as possible. In other words, each packet has to follow a path including at most two bends in the 3D case. [5] suggested that this lower bound may still hold even if the condition is removed; i.e., three-dimensional oblivious routing may be essentially inefficient. Fortunately this concern for 3D meshes was needless; we prove in this paper that any permutation can be routed over the 3D meshes including N processors in $1.16\sqrt{N} + o(\sqrt{N})$ steps by relaxing the condition a little bit: If we only allow the path of every packet to make one more bend, then the running time of the algorithm decreases from $\Omega(N^{2/3})$ to $\Theta(\sqrt{N})$. This upper bound is optimal within constant factor by [6] and does not change if we add the shortest-path condition.

For oblivious routing, there is a general lower bound, i.e., $\sqrt{N}\,/\,d$ for degree-d networks of any type [6]. This is tight for the hypercube including N processors, namely, $\Theta(\sqrt{N}/\log N)$ is both upper and lower bounds for oblivious routing over the hypercube [6]. This is also tight within constant factor for the 2D mesh including N processors, where $2\sqrt{N} - 2$ steps is an upper bound and also is a lower bound (without any supplementary condition) [1, 9, 10, 11]. Thus, tight bounds are known for two extremes, for the 2D mesh and for the $\log N$-dimensional mesh (= the hypercube). Furthermore, both upper bounds can be achieved rather easily, i.e., by using the most rigid, *dimension-order path strategy* [12, 2, 6]. However, for the 3D meshes, even the substantially weaker condition, i.e., the minimum-bending condition, provides the much worse bound as mentioned before [5]. Our second result now extends the family of meshes for which optimal oblivious routing is known. If randomization is allowed, then the bound decreases to $O(N^{1/3})$ [7, 13] for 3D meshes including N processors. A similar bound also holds for deterministic routing but for random permutations [9], including our new algorithm.

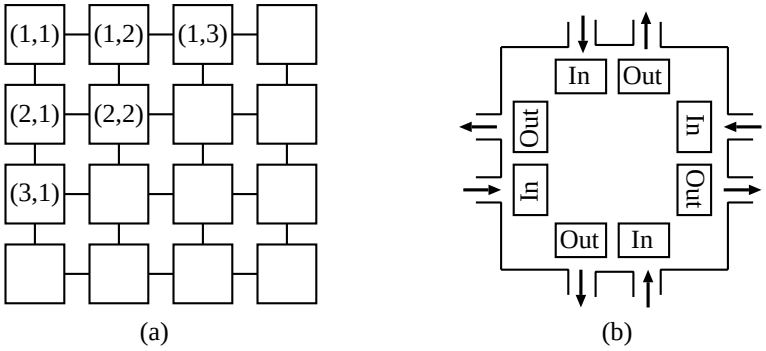


Figure 1: (a) 2D mesh (b) Processor

2 Models and Problems

Two-dimensional meshes are illustrated in Figure 1-(a). The following definitions on the two-dimensional (2D for short) mesh can be naturally extended to the three-dimensional (3D for short) mesh illustrated in Figure 2. A *position* is denoted by (i, j), $1 \leq i, j \leq \sqrt{N}$ and a processor whose position is (i, j) is denoted by $P_{i,j}$, i.e., the total number of processors is N. A connection between the neighboring processors is called a (*communication*) *link*. A *packet* is denoted by $[i, j]$, which shows that the destination of the packet is (i, j). (A real packet includes more information besides its destination such as its original position and body data, but they are not important within this paper and are omitted.) So we have N different packets in total. An *instance* of *permutation routing* consists of a sequence $\sigma_1\sigma_2\cdots\sigma_N$ of packets that is a permutation of the N packets $[1,1],[1,2],\cdots,[\sqrt{N},\sqrt{N}]$, where σ_1 is originally placed in $P_{1,1}$, σ_2 in $P_{1,2}$ and so on.

Each processor has four input and four output queues (see Figure 1-(b)). Each queue can hold up to k packets at the same time. The one-step computation consists of the following two steps: (i) Suppose that l (≥ 0) packets remain, or there are $k-l$ spaces, in an output queue Q of processor P_i. Then P_i selects at most $k-l$ packets from its input queues, and moves them to Q. (ii) Let P_i and P_{i+1} be neighboring processors (i.e., P_i's right output queue Q_i be connected to P_{i+1}'s left input queue Q_{i+1}). Then if the input queue Q_{i+1} has space, then P_i selects at most one packet (at most one packet can flow on each link in each time-step) from Q_i and send it to Q_{i+1}. Note that P_i makes several decisions due to a specific algorithm in both steps (i) and (ii). When making these decisions, P_i can use any information such as the information of the packets now held in its queues. Other kind of information, such as how many packets have moved horizontally in the recent t time-slots, can also be used. Note that it may happen that a packet does not go out of its source until a specific time in routing. In this paper, we assume that such an inactive packet

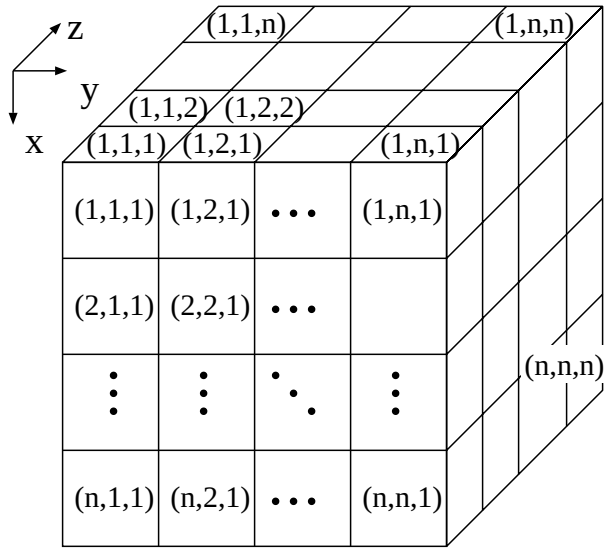


Figure 2: 3D mesh

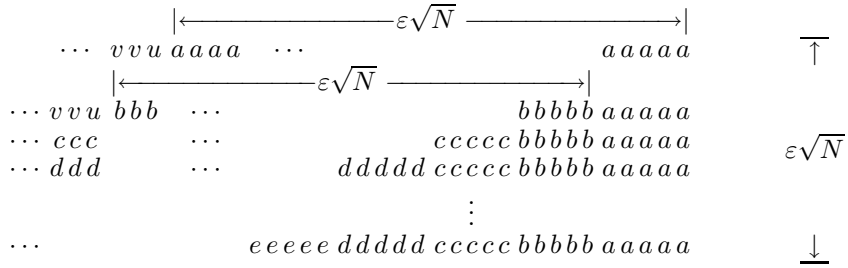
is not included in the queue-size.

If we fix an algorithm and an instance, then the path R of each packet is determined, which is a sequence of processors, $P_1(=$ source$), P_2, \cdots, P_j(=$ destination$)$. R is said to be *b-bend* if R changes its direction at b positions. A routing algorithm, A , is said to be *b-bend* if the path of every packet is at most b-bend. A is said to be *oblivious* if the path of each packet is completely determined by its source and destination. Furthermore, A is said to be *source-oblivious* if the moving direction of each packet only depends on its current position and destination (regardless of its source position). A is said to be *minimal* if the path of every packet is the shortest one. A is said to be *pure* if a packet never stays at the current position when it is possible for the packet to advance.

The most rigid and typical oblivious scheme for routing on meshes (and hypercubes) is the so called *dimension-order* algorithm. In the two-dimensional case, a packet first moves horizontally to its destination column and then moves vertically to its destination row. If the queue-size is unbounded, then the dimension-order algorithm can route any permutation in $2\sqrt{N} - 2$ steps on 2D meshes including N processors [9, 12]. However, if the queue-size is bounded above by some constant, then we have to pay great attention to algorithm's behavior, especially to the *queuing discipline*. As an example, let us consider the following oblivious routing algorithm, say, A_0, for $k = 1$, which is based on the dimension-order strategy, i.e., all packets move horizontally first and then make turns at most once at the crossings of source rows and destination columns. (i) Suppose that the top output queue of processor P_i is empty. Then

P_i selects one packet whose destination is upward on this column. If there are more than one such packet, then the priority is given in the order of the left, right and bottom input queues. (Namely, if there is a packet that makes a turn in this processor, then it has a higher priority than a straight-moving packet.) Similarly for the bottom, left and right output queues, i.e., a turning packet has a priority if competition occurs. (ii) If an input queue is empty, then it is always filled by a packet from its neighboring output queue. Thus each queue of the processor never overflows under A_0. It is not hard to see that A_0 completes routing within roughly $c\sqrt{N}$ steps for many "usual" instances. Unfortunately, it is not always true.

Consider the following instance: Packets in the lower-left one-fourth plane are to move to the upper-right plane, and vice versa. The other packets in the lower-right and upper-left planes do not move at all. One can see that A_0 begins with moving (or shifting) packets in the lower-left plane to the right. Suppose that the flow of those packets looks like the following illustration: Here a shows a packet whose destination is on the rightmost column, b on the second rightmost column and so on. Note that the uppermost row includes a long sequence of a's. The second row includes five a's and a long b's, the third row includes five a's, five b 's and long c's and so on. We call such a sequence of packets which have the same destination column a *lump* of packets.



Now the lump of a's reaches the rightmost column. One can see that the a's in the uppermost row can move into the vertical line smoothly and the following packets can reach to their bending position smoothly also: Thus nothing happens against the uppermost row. However, the packet stream in the second row will encounter two different kinds of "blocking:" (1) The sequence of five a's in the second row is blocked at the upper-right corner and cannot move upward since the $\varepsilon\sqrt{N}$ a's in the uppermost row have privileges. (2) One can verify that the last (leftmost) a of these five a's stops at the left queue of the second rightmost processor, which blocks the next sequence of b's, namely, they cannot enter the second rightmost column even if it is empty (see Figure 3). Thus, we need $\varepsilon\sqrt{N}$ steps before the long b's start moving. After they start moving, those b's in turn block the five b's on the third row and below. This argument can continue until the $\varepsilon\sqrt{N}$th row, which means we need at least $(\varepsilon\sqrt{N})^2$ steps only to move those packets.

One might think that this congestion is due to the rule of giving a higher priority to the packets that turn to the top from the left. This is not necessarily

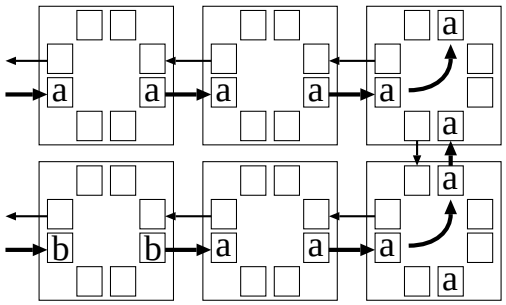


Figure 3: Blocking

true. Although details are omitted, we can create "adversaries" that imply a similar congestion against other resolution rules such as giving a priority to straight-moving packets. In the next section we will see how we can avoid this kind of bad blocking.

3 2D Oblivious Routing

3.1 Basic Ideas

Recall that the naive algorithm A_0 given in the previous section requires $\Omega(N)$ steps in the worst case. One can see that in the above example, serious delays do occur around at the crossings of source rows and destination columns, called the *critical zone*, and this inefficiency mainly comes from the co-existence of long and short lumps of packets; namely, short lumps of packets move ahead and prevent subsequent long lumps of packets from advancing. However, as shown in Lemma 3 later, the same algorithm runs quickly if there are only long lumps of packets in every row. On the other hand, if we have only short lumps of packets in every row, then we can also handle them efficiently (see Lemma 4). In the following, a lump of packets is said to be *long* if it includes at least d packets which have the same destination column for some positive integer d, which will be fixed later; otherwise, it is said to be *short*. So our basic strategy is quite simple: Before entering packets into their critical zone, (1) we first sort the sequence of packets in every row so that packets having nearer column destinations will go first (i.e., the packets move in nearest-first order), and then (2) count the size of each lump of packets which have the same destination column. As shown later, the operation (2) results in the sequence of the packets being changed into the reverse order. Note that it is only the purpose of these nearest-first and farthest-first orderings to count the number of packets going to the same column by grouping them. (3) We let only long lumps of packets go to the critical zone in farthest-first order. (4) The remaining short lumps of packets go afterwards but this time we adopt the so-called *shuffled order* as

the sequence of packets. For example, consider the following sorted sequence of packets:

$$d_2 \quad d_1 \quad c_3 \quad c_2 \quad c_1 \quad b_2 \quad b_1 \quad a_3 \quad a_2 \quad a_1$$

Then, the shuffle order sequence looks like the following illustration:

$$c_3 \qquad a_3 \qquad d_2 \qquad c_2 \qquad b_2 \qquad a_2 \qquad d_1 \qquad c_1 \qquad b_1 \qquad a_1$$

where a's have the farthest column destination, b's the second farthest, and so on as before. Namely, each of the rightmost four packets a_1, b_1, c_1 and d_1 comes from the right end of each lump, each of the next four packets a_2, b_2, c_2 and d_2 comes from the second right end of each lump and the remaining two packets are placed on the left end of the shuffle order sequence.

Recall that our main purpose of the sorting operation (1) is to gather packets heading for the same destination column, which makes it possible to execute the subsequent operations efficiently. Thus the key to implementing those ideas lies in designing an algorithm which can change any sequence of packets on each row to the sorted sequence according to the destination column without violating the oblivious condition:

Lemma 1 *Let $x = x_1 x_2 \cdots x_n$ be a sequence of n packets and $x_s = x_{s_1} x_{s_2} \cdots x_{s_n}$ be a sorted sequence of x such that x_{s_n} is the nearest packet among those n packets, and $x_{s_{n-1}}$ is the second nearest packet, and so on. Suppose that a linear array of $2n$ processors, P_1 through P_{2n}, is available and the sequence x of n packets is initially placed on the n processors of the left half of the linear array. Namely, P_1 through P_n hold x_1 through x_n in this order initially. Then there is an oblivious algorithm which runs in $2n-1$ steps and needs queue-size $k = 2$ such that the sorted sequence x_s is finally placed on the right half of the linear array, i.e., P_{n+1} through P_{2n} finally hold x_{s_1} through x_{s_n} in this order.*

Proof: The basic idea of the following oblivious algorithm is very similar to the idea implemented to adaptive routing in [4]: (i) We first move those n packets to the right in nearest-first order. That means the leftmost processor P_{n+1} of the right-half linear array receives packets in nearest-first order, i.e., P_{n+1} first receives the nearest packet x_{s_n}, next the second nearest packet $x_{s_{n-1}}$ and so on. (ii) Then we keep moving each packet up to its correct position. Here is a more detailed description:

(i) For $1 \le i \le n$, each P_i selects a packet which should go to the nearer column out of the packets it currently holds in its queue (an arbitrary one if all packets P_i holds have the same destination column), and moves it to the right at each step. However, P_i starts this action at the ith step and does nothing until then. If P_i has no packet, then it does nothing again.

(ii) For $1 \le i \le n$, each P_{n+i} simply shifts its packet sent from P_{n+i-1} to the right at each step. Then P_{n+i} eventually receives its correct packet x_{s_i} exactly at the $(2n-1)$th step.

Take a look at the following example:

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
initial positions	a	c	b	a	c	c	a	a	b	a

which shows that the sequence of 10 packets are now included in the left half, P_1 through P_{10}, of a linear array of 20 processors. Note that a should go to the farthest column, b the second farthest and so on as before. At the first step, only the leftmost packet, a in this example, moves to the right and P_2 now holds a and c in its input queue (recall that $k = 2$). In the second step, c is selected among a and c since c should go out earlier than a, and it goes to P_3. In the third step, P_2 sends the remaining packet a to the right, and at the same time, P_3 sends c out since the destination column of c is nearer than b. For example, after the eighth step, the positions change as follows:

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
after the 8th step						b	a	a	b	a
					a	a	c	c	c	

In the next step, P_{10}, P_9, P_8, P_7 and P_6 receive c, c, c, b and a from P_9, P_8, P_7, P_6 and P_5, respectively. In the next (10th) step, the packet c first goes out of the left half of the linear array and so on. The proof that the sorting operation works is based on the following claim, which can be shown by induction on m.

Claim 1 *For any $2 \leq m \leq n+1$, the following statement is true: At the $(m-1)$th step P_m receives the nearest packet among the packets initially placed on P_1 through P_{m-1}, at the mth step P_m receives the second nearest among those packets, and so on. Finally, at the $(2m-3)$th step, P_m receives the $(m-1)$th nearest (i.e., farthest) packet.*

Proof: See [4]. □

By Claim 1, P_{n+1}, the leftmost processor of the right half array, can receive the jth nearest packet $x_{s_{n-j+1}}$ exactly at the $(n+j-1)$th step for $1 \leq j \leq n$. In other words, P_{n+1} can receive the ith *farthest* packet x_{s_i} exactly at the $(2n-i)$th step for $1 \leq i \leq n$. Furthermore, the ith farthest packet x_{s_i} has to move rightward, and eventually, x_{s_i} arrives at its final position P_{n+i} exactly at the $(2n-1)$th step since each P_{n+i} simply shifts each of the packets sent from P_{n+i-1} to the right and x_{s_i} requires $(i-1)$ steps to travel from P_{n+1} to P_{n+i}. □

Note that the row routing algorithm in Lemma 1 works in linear time of n, but in order to obtain the sorted sequence of n packets we have to prepare a long path of (at least) $2n$ processors before entering the sequences into their critical zone.

3.2 Algorithms

Theorem 1 *There is an oblivious routing algorithm on the two-dimensional, $\sqrt{N} \times \sqrt{N}$ mesh of queue-size k ($2 \leq k \leq c\sqrt{N}$ for some constant c) which runs in $O(N^{0.75}/\sqrt{k})$ steps.*

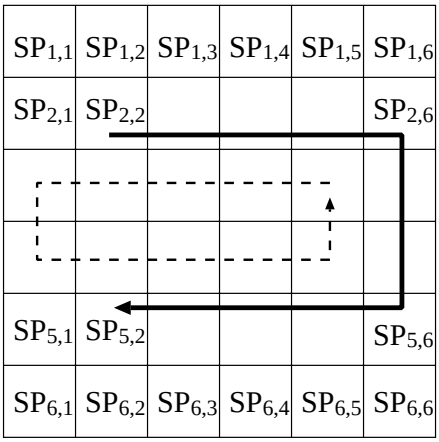


Figure 4: 36 subplanes

Proof: The whole plane is divided into 36 subplanes, $SP_{1,1}$ through $SP_{6,6}$ as shown in Figure 4. For simplicity, the total number of processors in 2D meshes is hereafter denoted by not N but $36n^2$, i.e., each subplane consists of $n \times n$ processors. (1) The entire algorithm is divided into 36×36 *phases*. In the first phase only packets whose sources and destinations are both in $SP_{1,1}$ move (i.e., they may be only a small portion of the n^2 packets in $SP_{1,1}$). In the second phase only packets from $SP_{1,1}$ to $SP_{1,2}$ move, and so on. (2) Suppose that it is now the phase where packets from $SP_{2,2}$ to $SP_{5,2}$ move. Then the paths of those packets are not shortest to make sorted sequences of the packets, but as shown by arrows in Figure 4: They first move rightward to $SP_{2,6}$, then downward to $SP_{5,6}$, and finally move leftward back to $SP_{5,2}$. Thus, the paths of the packets are denoted by the following sequence of subplanes, $SP_{2,2}$, $SP_{2,3}$, $\cdots$, $SP_{2,6}$, $SP_{3,6}$, $\cdots$, $SP_{5,6}$, $SP_{5,5}$, $\cdots$, $SP_{5,2}$. The general rule of path selection in each phase will be given soon. (3) These paths consist of three different zones, the (*parallel*) *shifting zone*, the *sorting zone* and the *critical zone*. In the above example, the sorting zone is composed of three consecutive subplanes $SP_{2,3}$, $SP_{2,4}$ and $SP_{2,5}$, the critical zone is $SP_{2,6}$ and the parallel shifting zone includes all the remaining portions.

We first describe the purpose of three different zones: First of all, before entering packets into the sorting zone, they move without changing their relative positions, i.e., they move in parallel just like a formation flight. In the above example again, packets initially placed on the top row in the source subplane $SP_{2,2}$ move through the top row without changing their order to the next subplane $SP_{2,3}$ of the sorting zone, packets on the second row in $SP_{2,2}$ move rightward through the second row without changing their order to $SP_{2,3}$, and so on. The sorting zone needs three consecutive subplanes where the flow of packets on each row is changed, i.e., from an arbitrary order to the farthest-first

order. More precisely, packets on each row in $SP_{2,3}$ of the sorting zone once move rightward to the next subplane $SP_{2,4}$ while sorting their order into the nearest-first order by using the row routing algorithm in Lemma 1. Furthermore, the packets move from $SP_{2,4}$ to the next subplane $SP_{2,5}$, while changing their order into the reverse order (i.e., farthest-first order) to calculate the size of each lump of packets, by using a new (but simple) idea described later. The critical zone is the most important zone where each packet enters its correct column position, but relatively within the subplane. Namely, if the destination of the packet is on the leftmost column in the final destination subplane $SP_{5,2}$, then the packet temporally enters the leftmost column in the critical zone $SP_{2,6}$, if the destination is on the second leftmost column in $SP_{5,2}$, then the packet temporally enters the second leftmost column in $SP_{2,6}$, and so on. Finally, all the packets move through the shifting zone towards their final positions without changing their relative positions.

Recall that our goal is to reduce path-congestion in the critical zone with a help of the sorting zone. To prepare the sorting zone before entering packets into the critical zone, the path of each packet from $SP_{2,2}$ to $SP_{5,2}$ takes such a long way. In general our oblivious routing algorithm has to determine the path independently for each of the 36^2 phases. Here is the rule: If the starting subplane is in the left side of the whole plane (as $SP_{2,2}$), then the packets are moved to the right end in the first stage of the algorithm, which allows us to prepare the sorting zone. If the starting subplane is on the right side, then the packets are moved to the left end in the first stage. If the starting and ending subplanes are on the same row, then the path goes like the one given by the dotted line in Figure 4. It should be noted that this path design is obviously source-oblivious in each time-step, but is not in the whole time-steps. To make it source-oblivious in the whole steps is not hard, see the remark at the end of this section.

Now we give more detailed description of a single phase of the algorithm Rout[2]. The size of each queue is two ($k = 2$) for a while, but it is extended to Rout[k] later. Suppose for exposition that it is now the phase where packets from a subplane in the left side of the upper half plane to another subplane in the lower half plane should move. We call those packets *active packets* on this phase. Also, suppose that the active packets move first through subplanes $PZ_1, \cdots, PZ_p$ as the parallel shifting zone, then move through subplanes SZ_1, SZ_2 and SZ_3 as the sorting zone, next move through subplane CZ as the critical zone, and finally move through $PZ'_1, \cdots, PZ'_q$ as the parallel shifting zone again for $p \geq 0$ and $q \geq 0$. (By $p = 0$ ($q = 0$) we mean that the algorithm needs no shifting subplanes before the sorting zone (after the critical zone).)

Algorithm: Rout[2]

A single phase of Rout[2] is divided into the following seven stages:

Stage 1: If $p = 0$, then go to Stage 2. Otherwise, the following is executed on every row in parallel: All the active packets in PZ_1 are routed horizontally into SZ_1 without changing their relative positions within every subplane. More

precisely, at each step, each processor shifts rightward its packet it currently holds in its queue. Eventually, all the active packets simultaneously arrive at their temporal destinations in SZ_1 exactly at the pnth step since it is exactly pn positions long for every active packet to travel to its temporal destination in SZ_1.

Stage 2 (Sorting): The following is executed on every row in parallel: The same algorithm as the one described in the proof of Lemma 1 is executed by using $2n$ processors, P_1 through P_{2n}, on the row in SZ_1 and SZ_2.

However, the present situation is a little bit different. Now some processor may have no packet since only active packets on this phase have been routed here. So, the initial sequence of packets on P_1 through P_n may include spaces or "null packets," but it does not cause any problem by regarding the null packets as the farthest (going out latest) packets.

Stage 3 (Counting): The following is executed on every row in parallel: Suppose that P_{n+1} through P_{3n} are $2n$ processors in the row in SZ_2 and SZ_3. Then we change the current position of each packet in SZ_2 into the symmetrical position with respect to the boundary between SZ_2 and SZ_3 as follows: (i) Each of the processors P_{n+1} through P_{2n} in SZ_2 simply shifts every active packet one position to the right step by step. (ii) Also, at each step, each P_{2n+i} in SZ_3 shifts rightward its packet it currently holds in its queue except for the packet, say, y_{2n+i} which arrives there exactly at the $(2i-1)$th step (from the beginning of this stage) for $1 \leq i \leq n$. Simultaneously, each P_{2n+i} calculates how many packets whose destinations are the same column as its packet y_{2n+i} it has sent to the right. Then, at the $(2n-1)$th step of this stage, P_{2n+i} stores the number of the packets plus one (for y_{2n+i}) in N_{2n+i}.

Stage 4 (Moving Long Lumps in Farthest-First Order): The following is executed on every row in parallel: If $N_{2n+i} \geq d$, then every P_{2n+i} in SZ_3 $(1 \leq i \leq n)$ starts to move its active packet to the right at the first step. Then the packet keeps moving one position to the right at each step. Otherwise, the processor does not start and remains holding its packet in its queue. However, if each P_{2n+j} with $N_{2n+j} < d$ receives a packet whose destination is the same column as its packet y_{2n+j} during this stage, then it starts to move y_{2n+j} to the right by using the farthest-first contention resolution protocol.

Stage 5 (Moving Short Lumps in Shuffle Order): The following is executed on every row in parallel: Note that there remain only packets of short lumps in SZ_3. At the first step, all P_{2n+i}'s such that the values of N_{2n+i}'s are equal to one start to forward their active packets to the right (if they have the packets in their queues), and then those packets keep moving rightward. At the $(n+1)$th step, all P_{2n+i}'s with $N_{2n+i} = 2$ start to move their packets to the right, at the $(2n+1)$th step all P_{2n+i}'s with $N_{2n+i} = 3$ start to move their packets to the right, and so on. Namely, n steps are inserted between the first actions. Finally, at the $((d-2)n+1)$th step, every P_{2n+i} with $N_{2n+i} = d-1$ starts to forward its packet. All the packets keep being forwarded to the right.

Stage 6 (Critical Zone): Right after the sorting zone, the packets enter the critical zone. Recall that long lumps enter first in farthest-first order and then short lumps in shuffle order. Each packet changes the direction from row to column at the crossing of its correct destination column (relatively in the subplane). What Rout[2] does in this zone is exactly the same as A_0, i.e., it gives a higher priority to turning packets.

Stage 7 (Shifting Zone): All active packets move in the shifting zone towards their final goals without changing their relative positions within the subplane if $q \geq 1$. (Otherwise, the packets should have already been routed to their final positions).

Stage 1 and Stage 2 take pn steps and $(2n-1)$ steps, respectively, as shown before. Suppose for example that $n = 10$ and the sorted sequence of 10 packets be now placed on processors P_{11} through P_{20} in SZ_2 after Stage 2 (see the proof of Lemma 1 again):

$$\begin{array}{cccccccccccccccccccc} P_{11} & P_{12} & P_{13} & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} & P_{19} & P_{20} & P_{21} & P_{22} & P_{23} & P_{24} & P_{25} & P_{26} & P_{27} & P_{28} & P_{29} & P_{30} \\ a_5 & a_4 & a_3 & a_2 & a_1 & b_2 & b_1 & c_3 & c_2 & c_1 & & & & & & & & & & \end{array}$$

One can see that if every packet moves one position to the right step by step in Stage 3, then c_1 arrives at the temporal destination P_{21} in the first step, c_2 arrives at P_{22} in the third step, c_3 arrives at P_{23} in the fifth step, and so on. Finally, the leftmost packet a_5 arrives at the rightmost processor P_{30} in the 19th step of this stage (since $2 \times 10 - 1 = 19$):

$$\begin{array}{cccccccccccccccccccc} P_{11} & P_{12} & P_{13} & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} & P_{19} & P_{20} & P_{21} & P_{22} & P_{23} & P_{24} & P_{25} & P_{26} & P_{27} & P_{28} & P_{29} & P_{30} \\ & & & & & & & & & & c_1 & c_2 & c_3 & b_1 & b_2 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array}$$

Thus, after Stage 3, the stream of packets is changed into the farthest-first order. More importantly, each of P_{21} through P_{30} now knows how many packets in the same lump exist on its right side by counting the number of the flowing packets. For example, P_{22} knows another c exists in P_{23}, i.e., P_{22} sets $N_{22} = 2$.

Suppose for a while that $d = 4$, i.e., a lump is defined to be long if it includes four or more packets and only the lump of a's is long in the above example. At the first step of Stage 4, P_{26} and P_{27} start to move a_1 and a_2 to the right, respectively, since $N_{26} = 5$ and $N_{27} = 4$. Since P_{28}, P_{29} and P_{30} eventually receive a_1 (or a_2) from the left, they move their packets, a_3, a_4 and a_5 to the right. Thus all a's are routed rightward. In similar ways, all the packets of long lumps move forward. Note that, for example, P_{21} who does not know whether the lump of c's is long (since $N_{21} = 3$) does not initiate the move of its packet, and thus this short lump remains in SZ_3.

In Stage 5, `ROUT[2]` works as follows: At the first step of Stage 5, P_{25} and P_{23} start to forward b_2 and c_3, respectively, to the right since $N_{25} = N_{23} = 1$. Then, at the $(n+1)$th step, b_1 and c_2 start to move since $N_{24} = N_{22} = 2$, and finally c_1 starts to move at the $(2n+1)$th step since $N_{21} = 3$. Namely, we change the farthest-first sequence to the following shuffle order sequence (which

may include spaces between some two packets):

$$c_1 \qquad c_2 \qquad b_1 \qquad c_3 \qquad b_2$$

Now we shall investigate the time complexity of Rout[2]. The following discussions are only about what happens in the critical zone since the time needed for the sorting and shifting zones is linear in n. Suppose that a packet σ is now in the left input queue of a processor P_i, where σ is to enter the destination column by making a turn. Then it is said that σ is *ready to turn*. We first show a simple but important lemma, which our following argument depends on.

Lemma 2 *Suppose that the rightmost packet of a lump L of packets is now ready to turn into its destination column. Then all the packets in L go out of the critical zone at most $3n$ steps regardless of the behavior of other packets.*

Proof: Suppose that the last (leftmost) packet α which is in the same lump L cannot move on. Then there must be a packet, β, which is ready to turn into the same column in some upper position of α and move on in the next time-step. Let us call such a packet β, *a blocking packet against* α. Note that β cannot be a blocking packet against α any longer in the next step. Thus, in each step from now on, the following (i) or (ii) must occur: (i) The leftmost packet α in L moves one position. (ii) One blocking packet against α disappears. Since there are at most n packets that are to turn into a single column, (ii) can occur at most n times. (i) can occur at most $2n$ times also since we have at most $2n$ processors on the path of each packet in the critical zone. □

Recall that a lump is said to be long if it includes at least d packets for some positive constant d; otherwise, it is said to be short.

Lemma 3 *All the long lumps can go through the critical zone within $O(n^2/d)$ steps.*

Proof: The following argument holds for any row: Let L_i be a lump of packets in some row such that packets of L_i head for a farther column than packets of L_j for $i < j$. Since packets of long lumps move in the farthest-first order, they flow in the following form of lumps:

$$\cdots L_3 \cdots L_2 \cdots L_1$$

Since each long lump has at least d packets, there are at most n/d different lumps in each row. Now the rightmost packet of L_1 must be ready to turn within $2n$ steps. By Lemma 2, then, those packets in L_1 must go out of the critical zone within the next $2n$ steps, i.e., within $4n$ steps in total. After L_1 goes out of the critical zone, L_2 must go out within $4n$ steps and so on. Thus $4n \times n/d = 4n^2/d$ is the maximum amount of steps all the long lumps need to go out. □

Lemma 4 *All the short lumps can go through the critical zone within $O(dn)$ steps.*

Proof: The flow of packets on each row looks as follows:

$$\cdots\ z_1\ \cdots\ z_{l-1}\ z_l\ y_1\ \cdots\ y_{j-1}\ y_j\ x_1\ \cdots\ x_{i-1}\ x_i$$

Here $x_1 \cdots x_i$ is a sequence of different packets such that x_i is heading for a farther column than x_{i-1}. Let us call this sequence an *ordered string*. $y_1 \cdots y_j$ is the next ordered string (i.e., y_j is heading for a farther column than x_1) and so on. One can see that each packet in the first ordered string becomes ready to turn within $2n$ steps and must go out of the critical zone within the next $2n$ steps by Lemma 2 (regardless of possible spaces between some two packets). After that, the second ordered string must be ready to turn within $2n$ steps and then goes out as before. Note that we have only d ordered strings since each short lump has less than d packets. Hence, $4n \times d$ is the enough number of steps before all the short lumps go out. □

Now we are almost done. When moving packets from the sorting zone to the critical zone, we first move only lumps whose size is at least $\sqrt{n}$. Then by Lemma 3, the routing for these "long" lumps will finish in $O(n\sqrt{n})$ steps. After that we move "short" lumps, which will be also finished, by Lemma 4, in $O(n^2/\sqrt{n}) = O(n\sqrt{n})$ steps.

Algorithm Rout[2] can be extended to Rout[k] for a general queue-size k: Suppose from now on that a lump is said to be long if it includes at least kd packets for some positive constant d; otherwise, it is said to be short. Then we can obtain similar lemmas, say Lemmas 2', 3' and 4' to Lemmas 2, 3 and 4, respectively for the general queue-size. Lemma 2' is exactly the same as Lemma 2. As for Lemma 3', we replace d by kd. Then the time complexity for Rout[k] is $O(n^2/kd)$. As for Lemma 4', we also replace d by kd and furthermore each ordered string $x_1 \cdots x_{i-1}x_i$ is replaced by $X_1 \cdots X_{i-1}X_i$. Here X_{i-m} consists of k packets of the same destination instead of only one packet for x_{i-m}. Since our queue-size is k this time, each X_{i-m} can fit into a single input queue, which means all the packets in X_{i-m} can get ready to turn within $2n$ steps. Also note that the number of the ordered strings is at most d. Hence the bound in Lemma 4, $O(nd)$, does not differ in Lemma 4', which is a benefit of the large queue-size.

If we set $d = \sqrt{n/k}$, then both $O(n^2/kd)$ and $O(nd)$ become $O(n^{1.5}/\sqrt{k})$. □

Remark. Recall that the current movement of packets is not source-oblivious in the whole time-steps, which can be modified as follows: Suppose that the goal subplane is in the left-lower of the whole plane. Then all the packets but in the rightmost subplanes move upward first. Then they move to the right in the uppermost subplanes, and then go downward in the rightmost subplanes (here we place the sorting zone), and finally go to the left. (Exceptionally, packets in the rightmost-lower subplanes move to the left first and then follow the same

path as above.) Thus there is only one direction set in each subplane, i.e., we can design a source-oblivious algorithm. However, here is one important point: Packets originally placed near the destination in this path cannot go through the sorting zone. Hence, we allow them once to go through the destinations and then join the standard path after that.

Remark. Extension to 3D case. Let the side-length of a 3D mesh be n (see Figure 2). Our algorithm consists of n sequential phases. In the first phase, only n^2 packets placed on the top horizontal plane are routed to their final positions. In the second phase, only n^2 packets placed on the second horizontal plane move, and so on. Here is an outline of each phase: (1) Each packet σ first moves up/down temporarily to the wth horizontal yz-plane if the destination of σ is on the wth vertical xy-plane. (2) The same algorithm as $\mathsf{Rout}[k]$ is performed on every horizontal plane in parallel, however, packet $[u, v, w]$ is routed temporarily to position (w, v, u). (3) Every packet moves along z-dimension to its correct z-coordinate and finally moves along x-dimension to its destination. Hence each phase can be performed in $O(n^{1.5}/\sqrt{k})$ steps ($2 \leq k \leq cn$ for some constant c) as follows: We need at most $n-1$ steps to move each of the n^2 packets placed on some horizontal plane up to its correct horizontal one, and we need $O(n^{1.5}/\sqrt{k})$ steps to move the packets in one horizontal plane. It is not hard to move each packet from the horizontal to the vertical planes in $O(n)$ steps. Since there are n sequential phases, the total routing time is $O(n \cdot n^{1.5}/\sqrt{k}) = O(n^{2.5}/\sqrt{k})$ (or $O(N^{5/6}/\sqrt{k})$ for $2 \leq k \leq cN^{1/3}$). Although details are omitted, the similar ideas may provide us some $o(N/\sqrt{k})$ upper bounds for oblivious routing on higher dimensional, constant queue-size meshes including N processors, e.g., an $O(N^{7/8}/\sqrt{k})$ upper bound for 4D meshes, an $O(N^{9/10}/\sqrt{k})$ upper bound for 5D meshes, and so on.

4 3D Oblivious Routing

For oblivious routing on any d-degree network, Kaklamanis *et al.* [6] proved an $\Omega(\sqrt{N}/d)$ lower bound. It is known [6, 9, 12] that this bound is tight for the hypercube and the 2D meshes. However, in the case of 3D meshes, it was not known whether the lower bound is indeed tight. Here is our answer:

Theorem 2 *There exists a deterministic, oblivious routing algorithm on the three-dimensional, $N^{1/3} \times N^{1/3} \times N^{1/3}$ mesh that routes any permutation in $1.16\sqrt{N} + o(\sqrt{N})$ steps.*

Remark. Theorem 2 holds for minimal and 3-bend routing. However, if we impose the 2-bend condition, then any oblivious routing algorithm requires $\Omega(N^{2/3})$ steps [5]. Recall that in the case of 2D meshes, dimension-order routing, which is even more rigid than the 1-bend routing, can achieve the tight $O(\sqrt{N})$ bound. Thus there is a big gap between 2D and 3D meshes in the difficulty of oblivious routing.

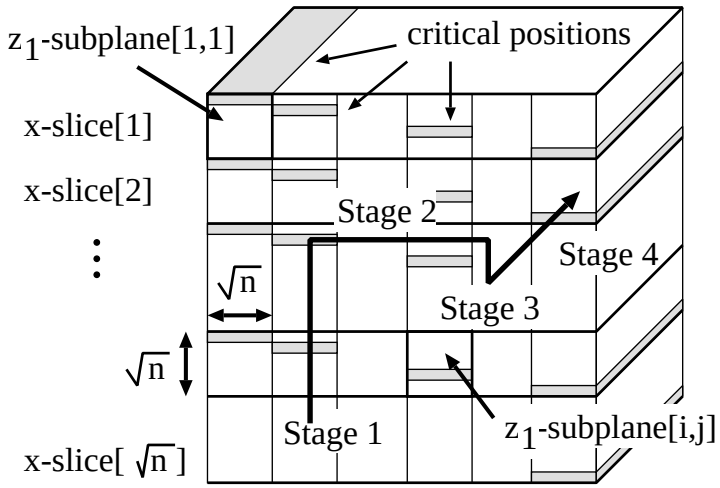


Figure 5: Slices, subplanes, critical positions

Proof: Let the total number of processors in 3D meshes be hereafter n^3, not N. We first present a deterministic, but non-minimal oblivious routing algorithm on the 3D mesh that routes any permutation in $O(n\sqrt{n})$ $(= O(\sqrt{N}))$ steps. At the end of this section, we will describe how to strengthen the algorithm to route every packet along its shortest path. See Figure 2 again. The three dimensions are denoted by *x-dimension*, *y-dimension* and *z-dimension*. *x_i-plane* $(1 \leq i \leq n)$ means the two-dimensional plane determined by $x = i$. Similarly for y_j-plane and z_k-plane. For example, the x_1-plane on the 3D mesh is the top, horizontal plane in Figure 2, which includes n^2 positions $(1,1,1), \cdots, (1,n,1)$, $(1,1,2), \cdots, (1,n,n)$. Let *x_iy_j-segment* $(1 \leq i,j \leq n)$ denote the linear array determined by $x = i$ and $y = j$, which consists of n processors $(i,j,1)$ through (i,j,n). Similarly for y_jz_k-segment and z_kx_i-segment.

The whole $n \times n \times n$ 3D mesh is partitioned into several submeshes. See Figure 5. *x-slice*[1] consists of the top $\sqrt{n}$ planes, the x_1-plane through the $x_{\sqrt{n}}$-plane. Similarly *x-slice*[2] consists of the next $\sqrt{n}$ planes. Each plane is divided into $\sqrt{n} \times \sqrt{n}$ groups, called *subplanes*. *z_k-subplane*$[i,j]$ on the z_k-plane $(1 \leq k \leq n)$ is the ith $\sqrt{n} \times \sqrt{n}$ 2D submesh from the top and jth from the left. Then $\sqrt{n}$ processors on the jth row from the top of each z_k-subplane$[i,j]$ are called *critical positions*, which play on an important role in the following algorithms. Note that the number of critical positions on a single z_kx_i-segment is exactly $\sqrt{n}$.

The following observation makes clear the reason why the elementary-path routing or dimension-order routing does not work efficiently: Suppose that n^2 packets placed on the z_1-plane should move to the x_1-plane. Also suppose that all of those n^2 packets first go up to the n processors on a single segment, $P_{1,1,1}$ through $P_{1,n,1}$, and then move along y-dimension. Then most of the n^2

paths probably go through one of the $n-1$ links between those n processors, which requires at least $\Omega(n^2)$ steps. Apparently we need to eliminate such heavy traffic on the single segment. The key strategy is as follows: The n^2 packets or paths (which go through a single segment in the worst case) are divided into $\sqrt{n}$ groups, $n\sqrt{n}$ paths each. Then the number of paths which a single segment has to handle can be reduced to those $n\sqrt{n}$ paths from the previous n^2 ones, where critical positions of each subplane play their roles. Here is our algorithm based on the dimension-order routing (see Figure 5 again):

Algorithm: DO-3-bend

Stage 1: Every packet moves along x-dimension to a critical position which is located in the same x-slice$[i]$ as its final destination. Namely, all of the packets go up or down into their correct x-slices.

Stage 2: Every packet placed now on the critical position moves along y-dimension to its correct y-coordinate, i.e., moves horizontally into its correct y_j-plane.

Stage 3: Every packet moves again along x-dimension to its correct x-coordinate, i.e., into its correct x_iy_j-segment.

Stage 4: Every packet moves along z-dimension to its final position.

One can see that the path of each packet is completely determined by its initial position and destination and that the algorithm can deliver all the packets for any permutation.

Lemma 5 DO-3-bend *can route any permutation in at most* $2n\sqrt{n} + o(n\sqrt{n})$ *steps.*

Proof: We shall analyze the running time of each stage: (1) Stage 1 requires at most n steps since every packet can move without delays. (2) Stage 2 requires at most $n\sqrt{n}+n$ steps. The reason is as follows: Recall that the number of critical positions on a single segment is exactly $\sqrt{n}$. Since each critical position holds at most n packets after Stage 1, the total number of packets that go through each z_kx_i-segment is at most $n\sqrt{n}$. Thus, it takes at most $n\sqrt{n}$ steps until those $n\sqrt{n}$ packets move out of the current subplane and it furthermore takes at most n steps until the packet which finally starts from the subplane arrives at its next intermediate position. (3) After Stage 2 every packet arrives at its correct sub-y_j-plane of the $\sqrt{n} \times n$ positions. Thus, the total number of packets that go through each link along x-dimension is at most $n\sqrt{n}$. Thus it takes at most $n\sqrt{n} + \sqrt{n}$ steps until all the packets arrive at their next positions. (4) Since each processor temporally holds at most n packets, $2n$ steps are sufficient for all the packets to arrive at their final positions. As a result, the whole algorithm requires at most $2n\sqrt{n} + o(n\sqrt{n})$ steps. □

The above algorithm is based on the dimension-order routing, i.e., all the packets move in the same direction in each stage. However, by making the

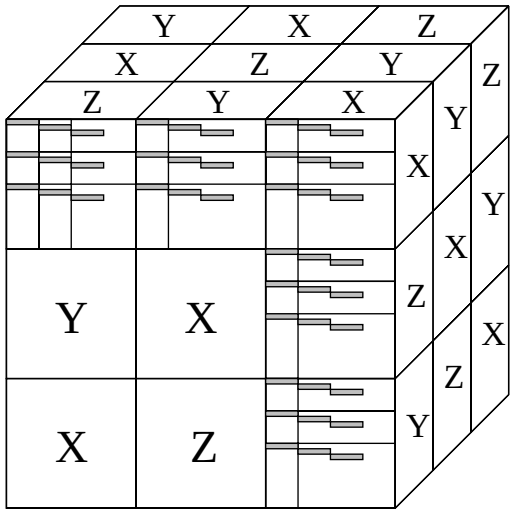


Figure 6: Three groups, X, Y, Z

packets move in different directions we can route more efficiently. The whole 3D mesh is partitioned into the three groups shown in Figure 6. Then, in the first stage, the algorithm NDO-3-bend moves all the packets which are initially placed on the groups marked "X", "Y" and "Z" along x-dimension, y-dimension and z-dimension, respectively. In the remaining stages, dimensions are switched adequately for each group. Note that each subplane can be viewed as a $\sqrt{n/3} \times \sqrt{n/3}$ 2D mesh. Now the following lemma holds:

Lemma 6 NDO-3-bend *can route any permutation in at most* $1.16n\sqrt{n}+o(n\sqrt{n})$ *steps.*

Proof: It also takes at most n steps for each packet to arrive at its critical position in the first stage. Note that there are $3 \times \sqrt{n/3}$ critical positions on a single $z_k x_i$-segment and each critical position holds at most $n/3$ packets after the first stage. Then it takes at most $n\sqrt{n/3}+n$ steps during the second stage. In the third and fourth stages, it takes at most $n\sqrt{n/3} + \sqrt{n/3}$ and $2n$ steps, respectively. Since $2/\sqrt{3} \leq 1.16$, the lemma holds. □

□

Finally we add the condition that all of the packets should follow their shortest routes: Suppose that a packet α which heads for the x_t-plane is originally placed on the x_s-plane for $s > t$. Then, in the first stage, α should go up along x-dimension to the closest but below critical position to the x_t-plane so as to follow its shortest path. Similarly for $s < t$. However, as a special case, if there is no critical position between α's original and destination planes, then α

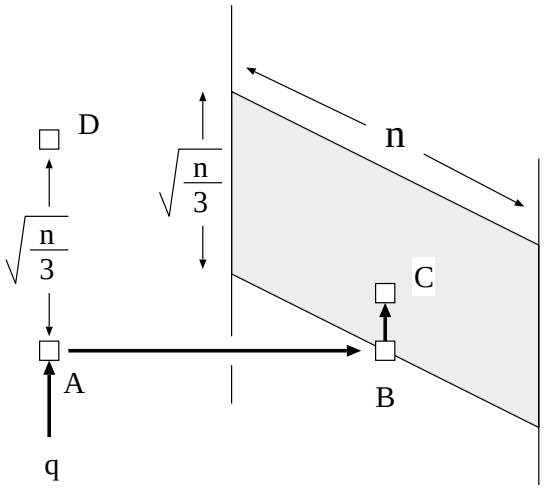


Figure 7: Minimal routing

does not move at all in the first stage. Let us count the total number of packets which temporally stay at each $z_k x_i$-segment (i.e., each horizontal segment) after the first stage. $\sqrt{3n}$ critical positions on a single segment hold at most $n\sqrt{n/3}$ packets, $n/3$ per critical position, and furthermore, at most n special packets may stay without moves. Thus it takes at most $n\sqrt{n/3}+2n$ steps in the second stage. For the third stage, at most $n\sqrt{n/3} + \sqrt{n/3}$ steps suffice again. The reason is as follows: See Figure 7. Suppose that a packet q first moves up vertically to a critical position, A, and then it moves horizontally to a position, B. Then q's destination must belong to the upper $\sqrt{n/3} \times n$ positions of B, which is painted gray in the figure; otherwise, q must first arrive at the upper critical position, D. This implies that the number of packets which go through the link between B and its upper neighbor C is bounded by $n\sqrt{n/3}$. Implementation of this idea is not hard:

Theorem 3 *There exists a deterministic, oblivious, minimal, 3-bend routing algorithm on the three-dimensional, $N^{1/3} \times N^{1/3} \times N^{1/3}$ mesh that can route any permutation in at most $1.16n\sqrt{n} + o(n\sqrt{n})$ steps.*

Remark. Each algorithm can be modified so as to satisfy the source-oblivious condition by increasing the constant factor of its running time slightly.

Remark. It is not hard to show that our algorithm runs for random permutations in $3n + o(n)$ steps with high probability. Here is the reason: Since there are $\sqrt{n}$ critical positions on each vertical segment, each critical position receives $\sqrt{n}$ packets on average in the first stage. Then $\sqrt{n} \times \sqrt{n} = n$ packets temporally stay on each horizontal segment. In the second stage, those n packets move horizontally in n steps, and each processor receives one packet on average.

The third stage requires $\sqrt{n}$ steps. Finally, all the packets arrive at their final positions in n steps.

Acknowledgments

The authors wish to thank two anonymous referees for providing detailed suggestions for improving our original paper.

References

[1] A. Bar-Noy, P. Raghavan, B. Schieber and H. Tamaki, "Fast deflection routing for packets and worms," In *Proc. 1993 ACM Symposium on Principles of Distributed Computing* (1993) 75-86.

[2] A. Borodin and J.E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *J. Computer and System Sciences* 30 (1985) 130-145.

[3] A. Borodin, P. Raghavan, B. Schieber and E. Upfal, "How much can hardware help routing?," In *Proc. 1993 ACM Symposium on Theory of Computing* (1993) 573-582.

[4] D.D. Chinn, T. Leighton and M. Tompa, "Minimal adaptive routing on the mesh with bounded queue size," *J. Parallel and Distributed Computing* 34 (1996) 154-170.

[5] K. Iwama and E. Miyano, "Three-dimensional meshes are less powerful than two-dimensional ones in oblivious routing," In *Proc. 5th European Symposium on Algorithms* (1997) 284-295.

[6] C. Kaklamanis, D. Krizanc and A. Tsantilas, "Tight bounds for oblivious routing in the hypercube," *Mathematical Systems Theory 24* (1991) 223-232.

[7] C. Kaklamanis, D. Krizanc and S. Rao, "Simple path selection for optimal routing on processor arrays," In *Proc. 1992 ACM Symposium on Parallel Algorithms and Architectures* (1992) 23-30.

[8] D. Krizanc, "Oblivious routing with limited buffer capacity," *J. Computer and System Sciences* 43 (1991) 317-327.

[9] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann (1992).

[10] F.T. Leighton, F. Makedon and I. Tollis, "A $2n - 2$ step algorithm for routing in an $n \times n$ array with constant queue sizes," *Algorithmica* 14 (1995) 291-304.

[11] J.F. Sibeyn, B.S. Chlebus and M. Kaufmann, "Deterministic permutation routing on meshes," *J. Algorithms* 22 (1997) 111-141.

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

[13] L.G. Valiant and G.J. Brebner, "Universal schemes for parallel communication," In *Proc. 1981 ACM Symposium on Theory of Computing* (1981) 263-277.