2次元メッシュ上での無情報ラウティング

宮野 英次
九州芸術工科大学
〒 815-8540 福岡市南区塩原 4-9-1
miyano@kyushu-id.ac.jp

岩間一雄
京都大学大学院情報学研究科
〒 606-8501 京都市左京区吉田本町
iwama@kuis.kyoto-u.ac.jp

あらまし:
2次元メッシュ計算機上での $O(\sqrt{N})$ 時間の無情報ラウティングアルゴリズムを示す．本モデルは，$\sqrt{N}\times\sqrt{N}$ 個のプロセッサを点対点結合した標準的なメッシュモデルであり，それぞれのプロセッサは，定数個までのパケットを一度に保持することができるキューを持つ．これまでに知られていた上限は $O(N^{0.75})$ である [IKM98].

キーワード： 2次元メッシュ，無情報ラウティング，上限

Oblivious Routing on 2-D Meshes

Eiji MIYANO
Kyushu Institute of Design
Fukuoka 815-8540, JAPAN
miyano@kyushu-id.ac.jp

Kazuo IWAMA
Department of Communications and Computer Engineering
Kyoto University
Kyoto 606-8501, JAPAN
iwama@kuis.kyoto-u.ac.jp

Abstract:
An $O(\sqrt{N})$ *oblivious* permutation-routing algorithm for 2-dimensional meshes is presented. The model is a standard mesh where $\sqrt{N}\times\sqrt{N}$ processors are connected via point-to-point connections and each processor has a queue which can hold only a constant number of temporal packets. The previous bound was $O(N^{0.75})$ [IKM98].

Key words: two-dimensional mesh, oblivious routing, upper bound

1 Introduction

Packet routing is clearly a fundamental problem in the area of parallel and/or distributed computing. Furthermore, among several variations of the routing problem, *permutation routing* has been most popular since it has been considered to be the best model to evaluate the overall efficiency of routing algorithms. The problem has a long history and there is a large literature. However, there are still several important unknowns; one of which was whether one can achieve an optimal time complexity by *oblivious strategy*, i.e., under the condition that the path of each packet is to be determined not depending on other packets. In this paper, we give an affirmative answer to this question.

Roughly speaking, routing is to determine each packet's path through the network by using various information, such as source addresses, destinations, and the configuration of network. The efficiency of a routing algorithm is generally measured by its *running time* and its *queue-size* of a processor, which is the maximum number of packets the processor temporally can hold at the same time during the routing. One of the most popular approaches is *adaptive* path selection. An adaptive algorithm is one in which the path a packet takes from its source to its destination may depend on other packets it encounters. For the meshes, there exist very efficient algorithms: Leighton, Makedon, and Tollis [LMT95] gave a deterministic algorithm with running time $2\sqrt{N}-2$, matching the network diameter, and constant queue-size. Sibeyn, Chlebus, and Kaufmann [SCK97] decreased the queue-size later. However, these algorithms are based on a kind of sorting and may be too complicated to implement on existence computers. If adaptive algorithms are limited to simpler ones, i.e., minimal, destination-exchangeable, and constant queue-size, then an lower bound jumps up to $\Omega(N)$ as proven by Chinn, Leighton, and Tompa [CLT96].

An *oblivious* path selection is another well-received approach [BH85, BRSU93, KKT91]. In the oblivious path selection, the entire path of each packet has to be completely determined by its source and destination before routing starts. Oblivious routing strategies generally make algorithms simple and hence the obliviousness may be a much desired property in practice. However, since contention resolution at each processor cannot change the moving direction of packets, it is quit tough problem to decrease both latency and queue-size simultaneously. Actually, several inefficiencies of oblivious routing are reported: A typical oblivious strategy for mesh network is the *dimension-order path* one. A packet first moves horizontally to its destination column and then moves vertically to its destination row. It is well known that in spite of very regular paths, the algorithm can route any permutation on the meshes in $2\sqrt{N}-2$ steps. However, unfortunately, a processor requires $\Omega(\sqrt{N})$ size queue in the worst case. In other words, if the queue-size is limited to some constant, then the algorithm must require much larger number of steps. Indeed, in the case of the bounded queue-size, Krizanc shows [Kri91] that *any* oblivious path selection on the meshes requires a much higher $\Omega(N)$ steps. This fact seems to imply that there does not exist a routing algorithm which is both efficient and practical.

However, more precisely, Krizanc's model is imposed much stronger restrictions on the normal oblivious condition; packet-scheduling has to be *pure*. The pure condition requires that each packet must move if its next position is empty. Little has been known whether the Krinzanc's $\Omega(N)$ bound can be improved by removing the pure conditions. Very recently, Iwama, Kambayashi, Miyano [IKM98] made a significant progress on this question: They gave an $O(N^{0.75})$ algorithm on two-dimensional, constant queue-size meshes by removing the pure condition. In this paper, we show that the $O(N^{0.75})$ bound can be further reduced to $O(\sqrt{N})$ by applying more refined packet-scheduling. Our algorithm controls over the flow of packets by using their column destinations and the queueing discipline to resolve link contention is sufficiently simple, i.e., only the *turning-packet-first* and *farthest-packet-first* rules are applied. This is the first deterministic oblivious algorithm that achieves $O(\sqrt{N})$ time in the constant queue-size case and our strategies may be capable of wide application. It also shows that the real reason for the very slow "$O(N)$" lower bound in [Kri91] was not the oblivious condition but the pure condition.

Under the randomized oblivious setting, Rajasekaran and Tsantilas [RT92] propose a simple ran-

domized algorithm which runs in $2\sqrt{N} + O(\log\sqrt{N})$ steps with high probability. However, the queue-size grows up to $\Omega(\log\sqrt{N}/\log\log\sqrt{N})$ large. In the same paper, they reduces the queue-size to some constant, unfortunately, at the sacrifice of the obliviousness. For three or more dimensional meshes, our knowledge is much less. For example, Iwama and Miyano shows an $\Omega(N^{2/3})$ lower bound in [IM97].

2 Our Models and Problems

Two-dimensional meshes are illustrated in Figure 1. 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 $P_{i,j}$. 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^2 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. 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 there remain l (≥ 0) packets, 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.

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$)$. A routing algorithm, 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 *pure* if a packet never stays at the current position when it is possible for the packet to advance.

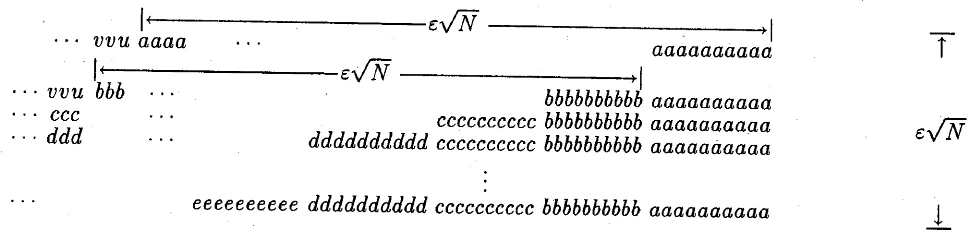
As an example, in the case where the queue-size is bounded to two, the dimension-order algorithm, say, A_0 for the meshes performs as follows: (i) If an input queue is empty, then it is always filled by a packet from its neighboring output queue. That is, A_0 is pure. (ii) Suppose that the top output queue of processor P_i has space only for one packet. Then P_i selects one packet whose destination is upward on this column. If there are more than one such packets, 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.

3 Previous Results

For pure and source-oblivious routing on the meshes, Krizanc [Kri91] gave an $\Omega(N)$ lower bound. On the other hand, Iwama *et al.* [IKM98] could design an oblivious routing algorithm which can route any permutation in $O(N^{0.75})$ steps by removing the pure condition. In this section, we point out how this algorithm broke Krizanc's linear lower bound.

We first observe why pure oblivious routing requires a lot of time. Consider the pure algorithm A_0 presented in the previous section and 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 ten a's and a long b's, the third row includes ten a's, ten 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.



The behavior of A_0 must be as follows: All lumps of a's reach the rightmost column at the same time. Then the a's in the uppermost row can move into the vertical line smoothly and the the following packets can reach to their bending position smoothly also: Thus nothing bad happens against the uppermost row. However, the packet stream in the second row will encounter two different kinds of "blocks:" (1) The sequence of ten a's is blocked at the upper-right corner since the $\varepsilon\sqrt{N}$ a's in the uppermost row have privileges. (2) One can verify that the last (leftmost) a of these ten 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. 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 ten b's on the third row and below. This argument can continue until the $\varepsilon\sqrt{N}$th row, which means A_0 requires at least $(\varepsilon\sqrt{N})^2$ steps only to move those packets.

What becomes possible if the pure condition is removed? – Suppose that a processor P first receives a packet a and then b from the left processor and sends both to the right. Without the pure condition, it is possible for P to keep a even if P's right input queue is empty. Hence P can send b first and then a to the right. Namely, the order of the flow of packets can be changed.

Now we seem to be able to route much quickly for the above instance as follows: As for the second row, we move the long lump of b's ahead of a's and turn b's into their column without any delay. Also as for the third row, the long c's move ahead of other packets and turn into their column smoothly, and so on. Actually, it is possible to execute this idea efficiently as shown in [IKM98]. The basic strategy of the $O(N^{0.75})$ algorithm proposed in [IKM98] is now clear: (i) The sequence of packets is divided into two groups, one consisting of only long lumps of *at least* $N^{0.25}$ packets and the other consisting of only short lumps of *at most* $N^{0.25}$ packets. (ii) The algorithm first routes only long lumps and then routes the remaining short lumps. The stages (i) and (ii) are performed in $O(\sqrt{N})$ and $O(N^{0.75})$ steps, respectively.

4 New Algorithm

4.1 Basic Ideas

As shown in the previous section, the naive oblivious routing algorithms requires a large number of steps in the worst case. However, on average, if the packets which should go to the same column are evenly distributed, then the algorithms can perform much better. Take a look at the following instance, which is similar to the previous example, but now packets of the same destination are distributed almost at random: The top row includes a large number of a's, a few b's and a few c's. The second row row includes a few a's and many b's, the third row includes a few a's and b's, and many c's and so on. As before, there is no delay in the top row. Also, everything will go well for the second row: The leftmost four b's reach to their

column and move up vertically. Subsequently, the next a reaches to its column but is blocked by the long a's of the top row. However, fortunately, this block does not do anything wrong. (i) The second leftmost b's also can move into their column smoothly. (ii) In a little while, the blocked a can advance since it finds a blank space in the top sequence of packets, the space which used to be occupied by another packet but a. Similar to the other rows. Thus serious delays should not occur.

$$\begin{array}{l} \cdots a\ a \cdots a\ b\ a\ a \cdots a\ c\ a\ a \cdots a\ b\ a\ a \cdots a\ a\ a\ a \\ \cdots a\ b\ b \cdots b\ c\ b\ b\ b\ a\ b\ b \cdots b\ c\ b\ b\ \cdots b\ a\ b\ b\ b\ b \\ \cdots c\ a\ c \cdots\cdots c\ b\ c\ c \cdots c\ a\ c\ c \cdots c\ b\ c \cdots c\ c\ c\ c \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \end{array}$$

So far we have observed the randomized sequence of packets. However, we can change any sequence of packets into the sequence such that packets of the same destination are uniformly distributed in a deterministic way. In the next section we will see how to achieve this, by inverse-sorting.

4.2 Inverse-Sorting

We introduce two operations, shuffle and inverse-sort, for sequence of packets. The inverse-sort is very similar to the *bit-reversal permutation* introduced in [Lei92].

Let $x = x_1x_2\cdots x_n$ and $y = y_1y_2\cdots y_m$ be sequences of packets. Then shuffle of x and y denoted by $SFL(x,y)$ is defined as follows:

$$SFL(x,y) = \begin{cases} x_1y_1x_2y_2\cdots x_my_mx_{m+1}\cdots x_n & \text{if } n \geq m \\ x_1y_1x_2y_2\cdots x_ny_ny_{n+1}\cdots y_m & \text{if } n < m. \end{cases}$$

For a sequence $x = x_1\cdots x_n$, its front half, x_F, is defined as $x_1\cdots x_{\lceil\frac{n}{2}\rceil}$ and its rear half, x_R, is $x_{\lceil\frac{n}{2}\rceil+1}\cdots x_n$. Then inverse-sort of string x, denoted by $SORT^{-1}(x)$, is defined as follows:

$$\begin{array}{ll} \text{If } |x| = 0, \text{ i.e., } x = \varepsilon, & SORT^{-1}(\varepsilon) = \varepsilon \\ \text{Otherwise,} & SORT^{-1}(x) = SFL(SORT^{-1}(x_F), SORT^{-1}(x_R)). \end{array}$$

Example 1.

$$SORT^{-1}(x_1x_2) = x_1x_2$$
$$SORT^{-1}(x_1\cdots x_4) = x_1x_3x_2x_4$$
$$SORT^{-1}(x_1\cdots x_8) = x_1x_5x_3x_7x_2x_6x_4x_8$$

Thus after the inverse-sort is applied, any subsequence in the original sequence $x_1x_2\cdots x_{16}$, say, $x_5x_6x_7$ is evenly distributed in the resulting sequence. Now the following lemma formally shows that inverse-sort realizes the desirable even distribution of packets.

Lemma 1. Let $x = x_1x_2\cdots x_n$ be a sequence and $z = x_ix_{i+1}\cdots x_{i+k-1}$ be its any subsequence of length k. Let x_{j_1}, x_{j_2} and x_{j_3} be any three symbols in z that appear in $SORT^{-1}(x)$ in this order. Then the distance between x_{j_1} and x_{j_3} is at least $\lceil\frac{n}{2k}\rceil$.

Proof. For simplicity we assume that n is a power of two.

Case 1. k is also a power of two and moreover i (= the starting position of z) is also a power of two. This case can be illustrated as in Figure 2, i.e., $z = z_2$. Let $\overline{z_i} = SORT^{-1}(z_i)$. Then, for example, $\overline{z_{23}} = SORT^{-1}(z_2z_3)$ can be written as $SFL(\overline{z_2}, \overline{z_3})$. This means one symbols in $\overline{z_3}$ is inserted between each pair of neighboring two symbols in $\overline{z_2}$. Now another two symbols are inserted when obtaining $\overline{z_{03}}$ and another four symbols are inserted when obtaining $\overline{z_{07}} = SORT^{-1}(z)$. Thus there are $\frac{n}{k}-1$ symbols inserted between the two symbols in z, namely, the distance between these two symbols in $SORT^{-1}(x)$ is $\frac{n}{k}$. This clearly satisfies the lemma.

Case 2. k is a power of two but i is not. See Figure 3. z can be written as $z_f z_r$ where z_f is a part of z_1 and a part of z_2. Among the these symbols x_{j_1}, x_{j_2} and x_{j_3}, two of them must come from z_f or z_r. The distance between those two symbols in $SORT^{-1}(x)$ is $\frac{n}{k}$ as described in Case 1. The distance between x_{j_1} and x_{j_3} is at least as large as this value.

Case 3. k is not a power of two. Now extend k into the minimal $k' > k$ that is a power of two and apply the argument in Case 2. Since $k > \frac{k'}{2}$, the distance between x_{j_1} and x_{j_3} is at least $\frac{n}{k'} > \frac{n}{2k}$. □

Lemma 2. Let $w = w_1 w_2 \cdots w_n$ be a sorted-sequence of n packets in increasing order according to the horizontal distance. Namely, w is the sequence such that the destination column of w_i is farther than or the same as the destination column of w_j for $i > j$. Then *any* sequence of n packets can be changed into $SORT^{-1}(w)$, called a ***completely-inverse-sorted sequence***, on a linear array of $2n$ processors of constant queue-size in $O(n)$ steps.

Proof. For better exposition, we describe the operation using a sequence $d_1a_1b_1a_2c_1b_2a_3c_2$ of eight packets and a linear array of 16 processor, P_1 through P_{16} (the sequence of packets may include spaces, but it does not cause any problem by regarding them as the null packets). Note that a should go to the farthest column, b the second farthest and so on as before. Also note that the sorted-sequence w and $SORT^{-1}(w)$ are as follows (see Example 1 again):

$$\begin{aligned} w &= d_1c_1c_2b_1b_2a_1a_2a_3 \ (= w_1w_2w_3w_4w_5w_6w_7w_8) \\ SORT^{-1}(w) &= d_1b_2c_2a_2c_1a_1b_1a_3 \ (= w_1w_5w_3w_7w_2w_6w_4w_8) \end{aligned}$$

Initially, the eight packets are placed on the left half of the 16 processors, one by one:

P_1	P_1	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
d_1	a_1	b_1	a_2	c_1	b_2	a_3	c_2								

Finally, the inverse-sorted sequence $SORT^{-1}(w)$ is included in the right half of the linear array (those destination may be temporary and each processor holds one packet):

P_1	P_1	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
								d_1	b_2	c_2	a_2	c_1	a_1	b_1	a_3

The basic idea is that (i) we first move those eight packets into the right half of the linear array in sorted-order, and then (ii) keep moving each packet up to its correct position. For example, a_2 is the fifth packet to enter into the right half and goes to P_{12}. It is known that the first idea (i) can be implemented in $O(n)$ steps [IKM98]. The idea (ii) is much easier: One can verify that for example a_2 which is now currently placed on P_9 arrives at the temporal destination P_{12} in third steps further, i.e., at the 12th step. As another example, c_2 arrives at P_9 in the 13th step, and finally at P_{11} in the 15th step. Similarly to other packets. One can see that (ii) can be realized in $O(n)$ steps. □

4.3 The Whole algorithm

The entire plane is divided into 16 subplanes, $SP_{1,1}$ through $SP_{4,4}$ as shown in Figure 4. For simplicity, the total number of processors in 2D meshes is hereafter denoted by not N but $16n^2$, i.e., each subplane consists of $n \times n$ processors.

The algorithm consists of 16×16 *phases*. In the first phase only packets whose sources and destinations are both in $SP_{1,1}$ move. In the second phase only packets from $SP_{1,1}$ to $SP_{1,2}$ move, and so on. Here is an outline of each phase: Suppose that it is now the phase where packets from $SP_{3,2}$ to $SP_{2,4}$ move. (i) They first move to $SP_{4,2}$, i.e., first move into the temporal subplane which is two subplanes away from the destination subplane both horizontally and vertically; furthermore, without changing their relative positions. (ii) The order of the sequence of those packets can be changed into the completely inverse-sorted order in two consecutive subplanes $SP_{4,2}$ and $SP_{4,3}$, called the ***inverse-sorting zone***. (iii) The packets move

to $SP_{4,4}$, called the *critical zone*. The critical zone is the most important zone where each packet enters its correct column position (relatively within the subplane). (iv) Finally, they move towards their final subplane $SP_{2,4}$. The path of the packets are not shortest as shown by a line in Figure 4. Apparently no congestion as described before occurs in stages (i) and (iv). Our goal is to reduce the congestion in the critical zone with a help of the inverse-sorting zone.

Now we are ready to give more detailed description of a single phase:

Stage 1: The packets move to the inverse-sorting zone without changing their relative positions.

Stage 2 (Inverse-Sorting Zone): All the packets once stop moving at their temporal positions and resume moving through the inverse-sorting zone, two consecutive subplanes, $SP_{4,2}$ and $SP_{4,3}$ in the case of the above example. A sequence of packets on some row of $SP_{4,2}$ moves into the next $SP_{4,3}$, and finally, all the packets are placed on n^2 processors of $SP_{4,3}$ in completely-inverse-sorted order.

Stage 3 (Critical Zone): The packets enter the critical zone, however, we further need a new operation, *spacing*. Now the completely inverse-sorted sequence of packets are in $SP_{4,3}$:

$$\begin{array}{cccccccc} P_9 & P_{10} & P_{11} & P_{12} & P_{13} & P_{14} & P_{15} & P_{16} \\ d_1 & b_2 & c_2 & a_2 & c_1 & a_1 & b_1 & a_3 \end{array}$$

In the first step, only the leftmost packet a_3 starts moving to the right. In the fourth step, the second leftmost b_1 starts moving, in the eighth step the third a_1 starts moving, and so on. Namely, three spaces are inserted between any neighboring two packets. Each packet changes the direction from row to column at the crossing of its correct destination column (relatively in the subplane) but turning packets are given a higher priority.

Stage 4: All packets move towards their final goals without changing their relative positions.

4.4 Time Complexities

We shall investigate the time complexity of the above algorithm. Since Stages 1, 2 and 4 are apparently linear in n (or $O(\sqrt{N})$), the following discussions are only about what happens in the critical zone.

Fix a single phase and consider the inverse-sorting zones, SP_1 and SP_2, and the critical zone, SP_3. Suppose that the uppermost row of SP_3 includes k_1 α's, where α's show packets whose destinations are on the jth column from left in the critical zone, the second uppermost row includes k_2 α's and so on. Also, suppose that the completely-inverse-sorted sequence of packets on every row is now placed in SP_2.

Now consider a processor $P_{i,j}$ at the cross-point of the ith row and jth column. Note that, from Lemma 1, $P_{i,j}$ can receive at most two α's during some particular window Δ of $\frac{4\times n}{2k_i} = \frac{2n}{k_i}$ steps since three spaces are inserted between any neighboring two packets in the third stage of the algorithm. Then we will show that the total number of α's which $P_{1,j}$ through $P_{i-1,j}$ can receive during the window Δ is at most $\frac{2n}{k_i}-2$ and hence the two α's currently held in $P_{i,j}$ can move up during the window Δ, for some i $(1 \le i \le n)$.

Since some $P_{l,j}$ $(1 \le l \le i-1)$ can receive at most two packets in $\lceil \frac{4n}{2k_l} \rceil$ steps, the number of α's which the processor $P_{l,j}$ receives during Δ of $\frac{2n}{k_i}$ steps is at most

$$\left(\frac{2n}{k_i} / \left\lceil \frac{2n}{k_l} \right\rceil\right) \times 2 \le \frac{4n}{k_i} / \frac{2n}{k_l} = \frac{2k_l}{k_i}.$$

Hence, the total number of α's which $P_{1,j}$ through $P_{i-1,j}$ can receive is at most

$$\frac{2(k_1 + k_2 + \cdots + k_{i-1})}{k_i} \le \frac{2n}{k_i} - 2$$

since $k_1 + k_2 + \cdots + k_{i-1} \le n - k_i$ holds. The same argument can be applied for each j $(1 \le j \le n)$.

As a result, any delay does not happen in the critical zone and hence Stage 3 is also linear in n.

References

[BH85] A. Borodin and J.E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *J. Computer and System Sciences* 30 (1985) 130-145.

[BRSU93] A. Borodin, P. Raghavan, B. Schieber and E. Upfal, "How much can hardware help routing?," In *Proc. ACM Symposium on Theory of Computing* (1993) 573-582.

[CLT96] D.D. Chinn, T. Leighton and M. Tompa, "Minimal adaptive routing on the mesh with bounded queue size," *Journal of Parallel and Distributed Computing* 34 (1996) 154-170.

[IM97] 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.

[IKM98] K. Iwama, Y. Kambayashi and E. Miyano, "New bounds for oblivious mesh routing," In *Proc. 6th European Symposium on Algorithms* (1998) to appear.

[KKT91] C. Kaklamanis, D. Krizanc and A. Tsantilas, "Tight bounds for oblivious routing in the hypercube," *Mathematical Systems Theory 24* (1991) 223-232.

[Kri91] D. Krizanc, "Oblivious routing with limited buffer capacity," *J. Computer and System Sciences* 43 (1991) 317-327.

[Lei92] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann (1992).

[LMT95] 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.

[RT92] S. Rajasekaran and T. Tsantilas, "Optimal routing algorithms for mesh-connected processor arrays," *Algorithmica* 8 (1992) 21-38.

[SCK97] J.F. Sibeyn, B.S. Chlebus and M. Kaufmann, "Deterministic permutation routing on meshes," *J. Algorithms* 22 (1997) 111-141.

[Tom94] 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).

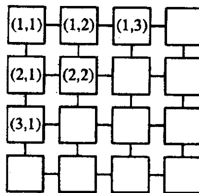


Figure 1: Two-dimensional mesh

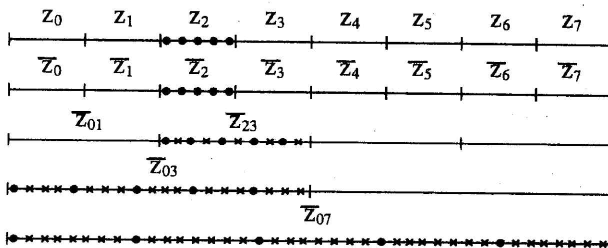


Figure 2: Case 1

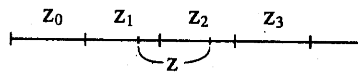


Figure 3: Case 2

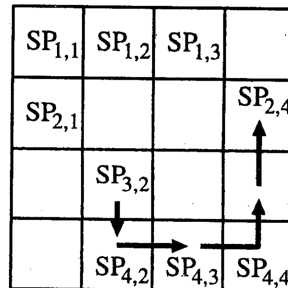


Figure 4: 16 subplanes