アルゴリズム 69－3
(1999. 9. 2)

メッシュ上でのビット反転置換を用いた ラウティングアルゴリズム

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:
本稿では，2次元，定数サイズキュー，$n \times n$ メッシュ計算機上でのラウティング問題に対する $(3+\varepsilon)n$ 時間の決定性無情報ラウティングアルゴリズムを示す．SODA99 において，我々は，$O(n)$ 時間の無情報ラウティングアルゴリズムが存在することを示していた．しかし，定数係数の具体的な値は示しておらず，その値は明らかに 1000 以上になっていた．

キーワード：2次元メッシュ，無情報ラウティング，ビット反転置換

Bit-Reversal Permutation Techniques for 2-D Mesh Routing

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

Kazuo IWAMA
School of Informatics
Kyoto University
Kyoto 606-8501, JAPAN
iwama@kuis.kyoto-u.ac.jp

Abstract:
In this paper we present an deterministic, oblivious, routing algorithm which runs in $(3+\varepsilon)n$ steps for any permutation routing on an $n \times n$ mesh using constant-size queues. In SODA99, the same authors showed that there is an $O(n)$ oblivious routing algorithm, but they did not give any concrete values for the leading constant factor, which is obviously more than 1000.

Key words: two-dimensional mesh, oblivious routing, bit-reversal permutation

1 Introduction

Mesh routing has received considerable attention for the last decades, and a variety of algorithms have been proposed. However, it is also true that there still remain several important unknowns. For example, until recently little had been known whether one can achieve an optimal, linear time bound for *oblivious* permutation routing on two-dimensional (2D) meshes of constant-size queues. In [2], Iwama and Miyano made a significant progress on this open question by giving an affirmative answer; they proposed a new technique, based on the *bit-reversal permutation*, to achieve an average scattering of packets, and by using this method they gave the first optimal (up to constant factor) algorithm for oblivious routing. Oblivious routing means that the path of each packet must be determined by its source and destination positions, which has been constantly popular since it makes routing algorithms significantly simple.

As for the queue-size, they need only two in [2], which is probably optimal. However, as for the time complexity, they only prove that their algorithm runs in linear time and it involves large constant factors partly due to making their proof on the time complexity clearer and simpler. Actually, the leading constant hidden in the big-O notation is at least more than 1000. Since no linear-time algorithms were known before, this must be an important progress theoretically. However, it is obviously questionable if this algorithm can claim much practical importance.

In this paper we focus mainly on the running time and present a $(3+\varepsilon)n$ oblivious algorithm which can route any permutation on 2D meshes for any small constant ε. Its queue-size is at most 12. The algorithm basically makes use of the bit-reversal permutation as before and its running time is around one and a half times as large as the network diameter of the meshes. Thus, this result shows that the oblivious algorithm based on the bit-reversal permutation does have practical merits and our new algorithm makes a major step toward absolutely optimal, $2n$-step algorithms with constant queue-size.

A typical (and popular in practice) oblivious strategy for mesh routing is called a *dimension-order algorithm*, which consists of two phases, the row routing and the column routing phases. Namely, 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 mesh in $2n-2$ steps. Unfortunately, however, some processor requires $\Omega(n)$-size queue in the worst case. To remove this bad path-congestion, the algorithm in [2] performs several operations, such as the bit-reversal permutation, against the sequence of packets. One drawback of this approach is that we need a long path of processors for these operations. To create such long paths, the algorithm in [2] suffers from serious detours and big constant factors. Fortunately, there is a standard technique to reduce this path length, i.e., simulating several processors by a single processor with a sacrifice of the queue-size (but still within some constant). This allows us to design a rather easy $6.5n$ algorithm. Our new $(3+\varepsilon)n$ algorithm needs a sequence of careful improvements from this one. One of them is a tighter analysis of the bit-reversal permutation, which doubles the rate of packet flow compared to the old algorithm.

Most previous algorithms are based on the *adaptive strategy*. In adaptive routing, the path of each packet from its source to destination may depend on other packets it encounters. For the meshes including $n \times n$ processors, the adaptive algorithms can provide very efficient time bounds: The first near-optimal algorithm was proposed by Kunde [4]; he showed that there is a deterministic algorithm with running time $2n + O(n/k)$ and queue-size k. Using Kunde's technique, Leighton, Makedon, and Tollis [7] gave a deterministic algorithm with running time $2n-2$, matching the network diameter, and constant queue-size. Rajasekaran and Overholt [9] and Sibeyn, Chlebus, and Kaufmann [11] decreased the queue-size later. However, all of these algorithms involve a flavor of mesh-sorting algorithms and may be too complicated to implement on existing computers. Thus, a simpler algorithm with a smaller queue-size might be of more practical interest, even if its running time is larger than $2n-2$.

There exist a lot of endeavors to simplify the routing schemes. Under the randomized oblivious setting, Valiant and Brebner gave a simple, randomized, oblivious routing algorithm which runs in $3n + o(n)$ steps with high probability. Rajasekaran and Tsantilas [10] reduced the time bound to $2n + O(\log n)$ later. However, the maximum queue-size grows up to $\Omega(\log n / \log\log n)$ large. By using another simple path selection, Kaklamanis, Krizanc, and Rao [3] gave a randomized algorithm with $2n + O(\log n)$ steps and constant-size queues, but at the sacrifice of the obliviousness. Under the adaptive setting, Chinn, Leighton, and Tompa [1] provided a *minimal*, adaptive routing algorithm which achieves $O(n)$ steps with constant size queue. However, in the same paper, they proved that if adaptive algorithms are limited to simpler ones, i.e., minimal, destination-exchangeable, and constant queue-size, then an lower bound jumps up to $\Omega(n^2)$.

In what follows, after giving a review of the $O(n)$ algorithm in [2], we first present the $6.5n$ algorithm in Section 3. Then we observe two major demerits of this algorithm and improves it into a $4n$ algorithm in Section 4. Finally we furthermore introduce technical improvements in Section 5, which achieves our final goal, i.e., the $(3+\varepsilon)n$ algorithm. We believe this step-by-step description helps for better exposition.

2 Models, Problems and Previous Algorithm

Our model in this paper is the standard, $n \times n$ mesh illustrated in Figure 1. 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 As shown in [6], if the queue-size is limited to some constant, then the dimension-order, greedy algorithm must require $\Theta(n^2)$ time in the worst case because heavy path-congestion may occur in the *critical positions*, where each packet changes its direction and enters its correct column. However, the following fact is also true: The greedy algorithm performs very well on average, i.e., if each packet has a random destination, then it can route all packets in $2n + O(\log n)$ steps with high probability as proven by Leighton [5]. This allows us to observe the congestion can also be avoided that if we can change an arbitrary sequence of packets into the sequence in such a way that packets of the same destination are almost evenly distributed. Thus Iwama and Miyano proposed the following new scheme: (i) Before routing those packets toward their destination, the order of packets in their flow is changed to control the injecting ratio of packets into the critical positions where serious delays can occur, by using the idea based on the *bit-reversal permutation.* (ii) Each packet move to its final destination. In the remaining of this section we shall give a brief overview of their key ideas.

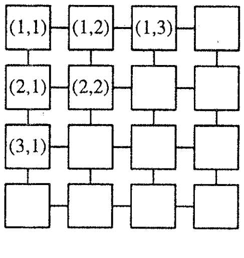

Figure 1: Two-dimensional mesh

Now we define the following two notations on sequences of packets on linear arrays:

Definition 1 (see e.g., [6]). Let $i_1 i_2 \cdots i_\ell$ denote the binary representation of an integer i. Then i^R denotes the integer whose binary representation is $i_\ell i_{\ell-1} \cdots i_1$. The *bit-reversal permutation* (BRP) π is a permutation from $[0, 2^\ell - 1]$ onto $[0, 2^\ell - 1]$ such that $\pi(i) = i^R$. Let $x = x_0 x_1 \cdots x_{2^\ell - 1}$ be a sequence of packets. Then the bit-reversal permutation of x, denoted by $BRP(x)$, is defined to be $BRP(x) = x_{\pi(0)} x_{\pi(1)} \cdots x_{\pi(2^\ell - 1)}$.

When $\ell = 3$, i.e., when $x = x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7$, $BRP(x) = x_0 x_4 x_2 x_6 x_1 x_5 x_3 x_7$. Namely, x_j is placed at the $\pi(j)$th position in $BRP(x)$ (the leftmost position is the 0th position).

Definition 2. For a sequence x of n packets, $SORT(x) = x_{s_0} x_{s_1} \cdots x_{s_{n-1}}$ denotes a sorted sequence according to the destination column, i.e., $SORT(x)$ is the sequence such that the destination column of x_{s_i} is farther than or the same as the destination column of x_{s_j} if $i > j$.

The following lemma plays a key role.

Lemma 1 [2]. Let $x = x_0 x_1 \cdots x_{n-1}$ be a sequence where $n = 2^\ell$ for some integer ℓ, and $z = x_i x_{i+1} \cdots x_{i+k-1}$ be its arbitrary subsequence of length k. Let x_{j_1}, x_{j_2} and x_{j_3} be any three symbols in z that appear in $BRP(x)$ in this order. Then the distance between x_{j_1} and x_{j_3} is at least $\lceil \frac{n}{2k} \rceil$.

Here is an important observation: Let $n = 2^\ell$ for some integer ℓ and suppose that a linear array of $2n$ processors, P_0 through P_{2n-1}, is available. Also suppose that a sequence $x = x_0 x_1 \cdots x_{n-1}$ of n packets is initially placed on the left half n processors of the linear array. Namely, P_0 through P_{n-1} hold x_0 through x_{n-1} in this order initially. Then it is known that we can obtain the sequence $BRP(SORT(x))$ of n packets by using these $2n$ processors, i.e., there is an algorithm which runs in linear time and which needs queue-size $K = 2$ and by which the sequence $BRP(SORT(x))$ is finally placed on the right half of the linear array, i.e., in P_n through P_{2n-1}. Take a look at the following sequence of n packets which has been already sorted in farthest-first order:

$$\cdots\ c_{k_3} \cdots c_2\ c_1\ b_{k_2} \cdots b_2\ b_1\ a_{k_1} \cdots a_2\ a_1.$$

Namely, $a_1, a_2 \cdots, a_{k_1}$ are the k_1 packets whose destinations are all on the rightmost column, $b_1, b_2, \cdots, b_{k_2}$ whose destinations are all on the second rightmost column, and so on. By Lemma 1, for example, any two neighboring packets among $a_{k_1}, \cdots, a_2, a_1$ are at least $\frac{n}{2k_1}/2 = \frac{n}{4k_1}$ positions apart in $BRP(SORT(x))$ on average. One can see that if k_1 packets of the same destination are truly even-distributed, then the distance should be $\frac{n}{k_1}$, or four times larger than what is guaranteed above. However, this can be compensated for easily if we do not have to care much for the constant of the time complexity. What we have to do is to insert three *spaces* between any neighboring two packets. Then the total length of the sequence becomes $4n$ and the distance between the two neighboring packets also becomes four times larger. Namely, by using the operation based on the bit-reversal permutation (and with the supplementary *spacing* operation), we can change the order of packets in their flow into the pseudo-random order which can remove serious delays at the critical positions. Thus we can achieve an $O(n)$ algorithm on meshes of constant queue-size. However, as mentioned above, we need $2n$ positions to change the order of n packets in their flow, i.e., we have to prepare the long permutation zone to operate the bit-reversal permutation. This is the main reason why the leading constant of the running time was large. In the next section we will see how to reduce this constant down to $6.5n$.

3 A $6.5n$ Algorithm

Recall that, in the previous section, we needed a linear array of $2n$ processors to obtain the sequence $BRP(SORT(x))$ of n packets. However, by simulating several processors with a single processor, the length of the linear array can be shortened with a sacrifice of the queue-size. The following lemma will be often used in the rest of the paper:

Lemma 2. For any positive constant, a linear array of n processors, P_0 through P_{n-1}, of queue-size K, can be simulated by a shorter linear array of $\lceil cn \rceil$ processors, Q_0 through $Q_{\lceil cn \rceil - 1}$, of queue-size $\lceil \frac{1}{c} \rceil \cdot K$.

Proof. This is a standard technique. See, e.g., [8]. □

For a while, we assume that the side-length n of meshes is denoted by 2^ℓ for some constant ℓ. The extension to the general length will be given at the end of the paper. Figure 2 illustrates how each packet moves from its source to destination. Roughly speaking, a packet whose source is in the left half and whose destination in the right half, denoted by a *LR-packets*, moves on the so-called simple path. The path of an LL-packet is a bit more complicated; it first goes to the left end of the row, changes its direction 180 degrees, returns to its correct column position and then goes to its final position. Similarly for RL-packets and RR-packets. One can see that the length of the path itself is at most $2n - 2$, which does not lose any optimality. Also, the path of each packet is not affected by other packets, i.e., the algorithm is oblivious.

However, how each packet moves on the path is not so simple. Figure 3 illustrates the left half of the single row (the right half is similar). LR-packets on this row are once "packed" into a small portion of length cn which is located at the right-end of the left half row. Here c is a small constant (≤ 0.125) and we call this small portion a *cn-tube*. LL-packets are similarly packed into another cn-tube located at the left-end of the row. Our first goal is to move those packets so that their original sequence x will be changed into $BRP(SORT(x))$ when they fit (in the sense of Lemma 2) in the cn-tube. It should be noted that the number of LR-packets in x can take any number between 0 and $n/2 - 1$. However, we consider that the length of x is always $n/2$, i.e., we assume that a null packet exists in the place which is originally occupied by an LL-packet. The packet movement till this moment is called *Stage 1*.

In *Stage 2*, each packet comes out of the cn-tube and moves to its correct column position. As described later we need to insert *spaces* between neighboring packets. In Stage 3, the packet moves on the correct column and finally reaches its destination.

Now look at each stage in more detail:

Stage 1 (Permutation). The path of an LR-packet p is actually a little different from the one

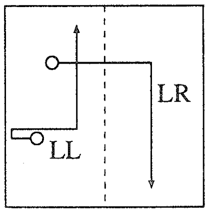

Figure 2: Paths of an LL-packet and an LR-packet

previously described if p is originally placed in the cn-tube. Now here is a lemma which gives the details of Stage 1:

Lemma 3. Let $x = x_0x_1 \cdots x_{n/2-1}$ be a sequence of LR-packets (possibly including null packets). Then there is an oblivious algorithm which moves x into the cn-tube where the order of the packets has been changed into $BRP(SORT(x))$. Furthermore, the running-time of the algorithm is $n-1$ and the maximum queue-size is $K = \lceil 1/c \rceil$ for some positive constant $c \leq 0.125$

Proof. The LR-packets originally placed in the right-side cn-cube are once squeezed out of the cn-tube or shifted exactly cn positions to the left. This shift operation is carried out along with the movement of LL-packets originally placed in the same cn-tube. Now the LR-packets stay P_1 through $P_{n/2-cn-1}$, at most one in each processor of P_1 through $P_{n/2-2cn-1}$ and at most two in each of $P_{n/2-2cn}$ through $P_{n/2-cn}$. Then we start the sorting operation by moving those LR-packets sequentially from the leftmost one. As one can see in a moment, we do not have to postpone the start of this sorting process until the shift is completed. Now here is a detailed description of the algorithm:

The sorting operation appears in [1, 8]. For $0 \leq i \leq n/2 - cn - 1$, P_i selects one packet which should go to the farther column out of the packets it currently holds, and moves it to the right at each step. However, P_i starts this action at the $(i+1)$th step and does nothing until then. If P_i has no packet, then it does nothing again.

The shift operation for LL-packets initially placed in the right-side cn-tube is as follows: At the first step, all the cn packets start moving leftward and keep being shifted one position to the left if they still need to move leftward. However, the following special care is required: Recall that LL-packets move leftward in the sorting phase in parallel, and those LL-packets have to move to the farther columns than the LR-packets, i.e., up to the left-end cn positions. Then if contention occurs, then we use the farthest-packet-first rule as the contention resolution rule and the LL-packet always has a higher

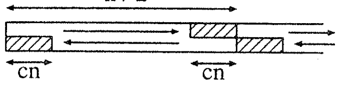

Figure 3: Packing packets into the cn-tubes

priority. By using the above sorting operation, at the cnth step the highest prioritized packet among the cn packets of the cn-tube moves out of there, at the $(cn+1)$th step the second highest packet among those packets moves out and so on. Eventually, at the $(2cn - 1)$th step, the cnth farthest (nearest) packet moves out of the cn-tube. This means that even if LL-packets are given a higher priority, all the LR-packets can move out of the cn-tube and, furthermore, each of them can arrive at its correct (temporal) position in $(2cn - 1)$ steps. Note that since at the next step P_{2cn-1} will start its first action of the sorting, $4cn$ should be at most $0.5n$, i.e., $c \leq 0.125$

Now we shall go back to the sorting. Since the farthest packet among LR-packets never delay, it enter into the right cn-tube at the $(n/2 - cn)$th step. Also, at the next step, the second farthest packet enter into the cn-tube, and so on. Finally, at the $(n-cn-1)$th step, the nearest packet arrives at $P_{n/2-cn}$. The bit-reversal permutation can be implemented in cn steps. As a result, the whole algorithm can be executed in $(n-cn-1)+cn = n-1$ steps and its queue-size is $\lceil 1/c \rceil$. $\square$

Stage 2 (Spacing): Recall that an LR-packet moves along the simple path, on the other hand, an LL-packet first goes to the left end of the row, changes its direction 180 degrees, returns to its correct column position. We further execute the spacing operation; seven spaces are inserted between any neighboring packets. Consider the leftmost cn processors P_0 through P_{cn-1} on some row. Now P_0 must hold the head $\lceil \frac{1}{c} \rceil$ LL-packets of the bit-reversally permuted sequence and P_1 must hold the next head $\lceil \frac{1}{c} \rceil$ packets and so on. In the first step, P_0 starts sending the head packet of the sequence to the right, and then the packet keeps moving one position to the right at each step. However, the other packets do not move at all during the first eight steps. In the ninth step, the second head packet starts moving, in the 17th step, the third head packet starts moving. Namely, seven steps are inserted between the first actions of any neighboring two packets. Then P_1 takes over the action, next P_2 starts, and so on, but every packet moves along its correct pat. Once each packet starts, it keeps being shifted one position to the right if it still needs to move rightward. The cn processors of the right-end cn-tube move their packets by using the same idea, but always move them rightward.

Each packet changes the direction from rightward to upward/downward at the crossing of its correct destination column. As the contention resolution rule, turning packets are always given a higher priority than straight-moving packets.

Stage 3 (Column routing): At each step each processor moves upward/downward its packet if it still needs to move.

Theorem 1. There is an oblivious routing algorithm on 2D meshes of queue-size eight which runs in $6.5n$ steps

Proof. We only need a large queue-size when we pack the packets into the cn-tube. Note that there is no congestion at the critical positions as shown below. Thus by setting $c \leq 0.125$, we can get the queue-size of eight by Lemma 3.

As for the running time, one should recall that the path-length of each packet is at most $2n$. The overhead occur when (i) the packets are packed and (ii) spaces are inserted. (Again no delays occur at the critical positions). (i) By Lemma 3, we need $n-1$ steps to pack the packets. If we would not need this packing operation, then the first packet would cross the middle of the mesh from left to right at time-step zero. Actually, the first packet cross there at time-step n. Thus the delay is $1.0n$. (ii) Seven spaces are inserted between neighboring two packets. Hence the last packet cross the middle in $4n$ steps since the first packet crossed there. Since the last packet would cross there in $0.5n$ steps if we would need no spaces. Thus the delay is $3.5n$ steps. In total we need $2n + n + 3.5n = 6.5n$ steps (at most) before routing is completed.

Now we shall prove that no congestion occurs at the critical positions. Suppose that the uppermost row includes k_1 α_1's, the second uppermost row includes k_2 α_2's and so on. Here α_i's are packets whose column destinations are the same, say, the jth column and move upward in the column. In general, the ith row includes k_i α_i's.

Now consider a processor $P_{i,j}$ at the cross-point of the ith row and jth column. It is important to note that, from Lemma 1, $P_{i,j}$ receives at most two α_i's during some particular window Δ_i of $\frac{2n}{k_i}$ ($\leq 8 \times \left\lceil \frac{n/2}{2k_i} \right\rceil$) steps since the length of the sequence is $n/2$ and seven spaces are inserted between any neighboring two packets in the second stage of the algorithm. If neither of those two packets α_i's can move up on at some step, then there must be a packet which is now ready to enter the jth column by making a turn at some upper position than $P_{i,j}$, and which can move on in the next time-step. Let us call such a packet *blocking packet against α_i's*. Note that the blocking packet against α_i's never block them again. In the following, we shall show that the total number of α_m's ($1 \leq m \leq i-1$) which $P_{1,j}$ through $P_{i-1,j}$ can receive during the window Δ_i is at most $\frac{2n}{k_i} - 2$. In other words, there must be at least two time-slots such that no packets flow on the jth column. Since there are no blocking packets at those time-slots, the two α_i's currently held in $P_{i,j}$ can move up during the window Δ_i.

Since $P_{m,j}$ can receive at most two α_m's in $\lceil \frac{2n}{k_m} \rceil$ steps, the number of α_m's which $P_{m,j}$ receives during Δ_i of $\frac{2n}{k_i}$ steps is at most

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

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

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

since $k_1 + k_2 + \cdots + k_{i-1} \leq n - k_i$. The same argument can apply for any j ($1 \leq j \leq n$) and for any window Δ_i ($1 \leq i \leq n$). As a result, any delay does not happen in the column routing phase. □

4 A $4n$ Algorithm

Recall that there are two different overheads in the $6.5n$ algorithm, $1.0n$ for due to packing and $3.5n$ due to spacing. In this section, we remove the latter delay. Unfortunately, it doubles the first delay and so the total delay will be 2.0 or the time complexity will be $4n$.

To remove the spacing delay, we introduce two measures. One is a tighter analysis of the bit-reversal permutation, which will given in Section 4.1. The other is more technical: Recall that there was the equation $k_1 + \cdots + k_{i-1} \leq n - k_i$ in the proof of Theorem 1. Reading carefully the argument there, one can see that if this equation can be changed to $k_1 + \cdots + k_{i-1} \leq \frac{n}{2} - k_i$, then the number of spaces can be decreased into three. This new equation means that the number of packets which can be blocking packets against α_i is at most $n/2$. This observation must be true if at most $n/2$ packets (instead of n packets currently) enter a single column. In Section 4.2, we divide the whole mesh into two portions, a upper half and a lower half. All packets whose destination is in the upper (lower) half are first moved to the upper (lower) half. Then routing is conducted independently in the upper and the lower halves, which meets our requirement mentioned above.

4.1 Tighter Analysis of Bit-Reversal Permutation

Recall that Lemma 1 says that the distance between any neighboring two packets which come from a subsequence of k packets is at most $\frac{n}{4k}$ on average. On the other hand, the distance is expected to be $\frac{n}{k}$ if the packets are truly distributed at random. This observation suggests that there is room for tighter analysis of the bit-reversal permutation than the previous one.

Lemma 4. Let $x = x_0x_1\cdots x_{n-1}$ be a sequence where $n = 2^\ell$ for some integer ℓ. Also let y_i be a subsequence of x such that $x = y_0y_1\cdots y_{\frac{n}{L}-1}$ where $|y_i| = L = 2^{\ell_1}$ for some integer $\ell_1 \leq \ell$. Namely, $y_i = x_{(i-1)L}x_{(i-1)L+1}\cdots x_{i\cdot L-1}$. Let x_{i_1} and x_{i_2} be any two symbols in y_i that appear in $BRP(x)$ in this order. Then the distance between x_{j_1} and x_{j_2} is exactly $\frac{n}{L}$.

Proof. Let $bin(j)$ be the binary representation of the integer j. Consider the subsequence $y_i = x_{(i-1)L}$ $x_{(i-1)L+1}\cdots x_{i\cdot L-1}$. Since $(i-1)L$ (= the leftmost position of y_i) can be divided by L and $|y_i| = L$, the lower ℓ_1 digits of $bin((i-1)L)$ are all 0 and the lower ℓ_1 digits of $bin(i\cdot L-1)$ are all 1. Furthermore, the upper $(\ell-\ell_1)$ digits of $bin((i-1)L), bin((i-1)L+1), \cdots, bin(i\cdot L-1)$ are all the same. That means that the lower $(\ell-\ell_1)$ digits of $bin(((i-1)L)^R), bin(((i-1)L+1)^R), \cdots bin((i\cdot L-1)^R)$ are all the same. It then follows that any two neighboring symbols among $x_{(i-1)L}, x_{(i-1)L+1}, \cdots, x_{i\cdot L-1}$ are exactly $2^{\ell-\ell_1} = \frac{n}{L}$ positions apart in $BRP(x)$. □

Lemma 5. Let $x = x_0x_1\cdots x_{n-1}$ be a sequence where $n = 2^\ell$ for some integer ℓ, and $z = x_ix_{i+1}\cdots$ x_{i+k-1} be its any subsequence of length $k \geq 13$. Let $x_{j_1}, x_{j_2}, \cdots, x_{j_{13}}$ be any 13 symbols in z that appear in $BRP(x)$ in this order. Then the distance between x_{j_1} and $x_{j_{13}}$ is at least $\lceil\frac{6n}{k}\rceil$.

Proof. Divide x into subsequences such that $x = y_0y_1\cdots y_{\frac{n}{L}-1}$ where $|y_i| = \frac{n}{L}$ and $L = 2^{\ell_1}$ for some integer $\ell_1 \leq \ell$. If z exactly coincide with $y_iy_{i+1}\cdots y_{i+k}$, then we can use the previous lemma. Otherwise, z starts at some middle position of y_i and ends also at some middle position of y_{i+k}.

We consider the following two cases due to the value of k, $2L < k \leq 3L$ and $3L < k \leq 4L$.

Case 1. In the case of $2L < k \leq 3L$, there are further two cases:

(1-1) z covers a part of y_i, y_{i+1}, y_{i+2}, and a part of y_{i+3}. From Lemma 4, any two neighboring symbols among y_{i+1} are exactly $\frac{n}{L}$ apart in $BRP(x)$. Similarly for y_{i+2}. Since a from part of y_i is included in z, the distance of any two neighboring symbols among y_i is more than $\frac{n}{L}$. By the similar reason, the distance of any two symbols among y_{i+3} is more than $\frac{n}{L}$. Thus some sequence of $\frac{n}{L}-1$ symbols in $BRP(x)$ includes at most one symbols of y_i. Also, the sequence includes at most one symbols of each of y_{i+1} through y_{i+3}. Since $k > 2L$, some sequence of $\frac{2n}{k} < \frac{n}{L}$ includes at most four symbols of z, i.e., the distance between any five symbols of z in $BRP(x)$ is at least $\frac{2n}{k}$. This means that the distance between x_{j_1} and $x_{j_{13}}$ is at least $3\cdot\frac{2n}{k} \leq \lceil\frac{6n}{k}\rceil$.

(1-2) z covers a part of y_i, y_{i+1}, and a part of y_{i+2}. By the same argument as above, some sequence of $\frac{2n}{k} < \frac{n}{L}$ includes at most one symbols of each of y_i, y_{i+1} and y_{i+2}, i.e., three symbols in total. Thus, the distance between x_{j_1} and $x_{j_{13}}$ is at least $4\cdot\frac{2n}{k} \leq \lceil\frac{8n}{k}\rceil$, which clearly satisfies the statement of the lemma.

Case 2. In the case of $3L < k \leq 4L$, there are also the following two cases:

(2-1) z covers a part of y_i, y_{i+1}, y_{i+2}, y_{i+3}, and a part of y_{i+4}. By Lemma 4, any neighboring two symbols are $\frac{n}{L}$ positions apart in $BRP(x)$. Similarly for y_{i+1}, y_{i+2}, y_{i+3}, y_{i+4}. Since some sequence of $\frac{3n}{k} < \frac{n}{L}$ in $BRP(x)$ includes at most five symbols of z, i.e., the distance between any six symbols of z in $BRP(x)$ is at least $\frac{3n}{k}$. This means that the distance between x_{j_1} and $x_{j_{11}}$, is at least $2\cdot\frac{3n}{k} \leq \lceil\frac{6n}{k}\rceil$, which satisfies the lemma.

(2-2) z covers a part of y_i, y_{i+1}, and a part of y_{i+2}. By the same argument as (2-1), some sequence of $\frac{3n}{k} < \frac{n}{L}$ includes at most one symbols of each of y_i, y_{i+1}, y_{i+2} and y_{i+3}, i.e., four symbols in total. Thus, the distance between x_{j_1} and $x_{j_{13}}$ is at least $3\cdot\frac{3n}{k} \leq \lceil\frac{9n}{k}\rceil$, which satisfies the lemma.

□

4.2 Removing Spaces

Before packing packets into the cn-tubes, we execute the following column routing:

Stage 0 (Column routing): All the packets which move from the upper half plane to the lower half plane are shifted exactly $\frac{n}{2}$ positions downward, one position at each step. Also, all the packets which move from the lower half plane to the upper half plane are shifted exactly $\frac{n}{2}$ positions upward, one position at each step.

This stage apparently takes at most $\frac{n}{2}$ steps, i.e., no overhead occurs at this moment. One can see that there are (at most) n packets in total on the left half of some row, at most two packets per each processor. In Stage 1, we pack those n packets into a cn-tube by using the similar sorting process. Unfortunately, however, a new $1.0n$ overhead is created for Stage 2. Although the farthest packet can enter its cn-tube at most in $0.5n$ steps, the nearest packet has to wait all the $n-1$ packets instead of $0.5n-1$ packets previously. Also, the last packet has to wait within the cn-tube until the other $n-1$ packets cross the middle instead of $0.5n-1$. However, the total overhead will be reduced: We can remove the spacing overhead greatly due to decrease of the total number of packets which may flow in a single column. By combining the tighter analysis of the bit-reversal permutation with this improvement, we can show that the spacing operation is no longer required:

Theorem 2. There is an oblivious routing algorithm on 2D meshes of queue-size 12 which runs in $(1+\varepsilon)n$ steps for any small constant ε.

Proof. Again consider a processor $P_{i,j}$ on the ith row and jth column and the same situation as the proof of Theorem 1. Then, from Lemma 5, $P_{i,j}$ receives at most 12 α_i's during new particular window Δ_i of $\frac{6}{k_i}$ steps. Similarly, the number of α_m's which $P_{m,j}$ receives during Δ_i is at most $12k_m/k_i$. Hence, the total number of α's which $P_{1,j}$ through $P_{i-1,j}$ can receive is at most

$$\frac{12(k_1+k_2+\cdots+k_{i-1})}{k_i} \leq \frac{6n}{k_i} - 12$$

since $k_1+k_2+\cdots+k_{i-1} \leq n/2-k_i$. Thus, 12 α_i's currently held in $P_{i,j}$ can move upward/downward during the window Δ_i and the spacing is no longer required. Since we can remove $3.5n$ overhead at the sacrifice of $1.0n$ overhead, the time complexity is reduced to $6.5n - 3.5n + 1.0n = 4.0n$ steps. □

5 A $(3+\varepsilon)n$ Algorithm

Now we are suffering from $2n$ overhead for the packing operation, which will be reduced into $(1+\varepsilon)n$ in this section. The basic idea is as follows: See Figure 4 which illustrates the left half of some row. Let x_1 and x_2 be the sequence of LR-packets after Stage 2 of the $4n$ algorithm (each processor may have two packets). Previously, x_1 and x_2 are sorted as a one sequence, which created roughly n overhead since the leftmost packet may have to wait all the other packets. This time we sort x_1 and x_2 *in parallel* and then merge them. More in detailed, x_1 is sorted and into T' which is a similar tube as T but placed in the left-end quarter. x_2 is also sorted into T. Then we can merge these two sequences as moving them to the right by using the similar sorting process, where we use the rank of each packet within the whole bit-reversal permutation as key. By this, the overhead can be roughly valued and this reduction of overhead can farther be achieved by increasing the degree of parallelism.

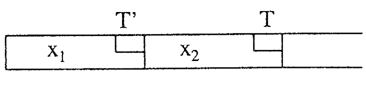

Figure 4: Parallel sort

Theorem 3. There is an oblivious routing algorithm on 2D meshes of queue-size 12 which runs in $(1+\varepsilon)n$ steps for any small constant ε.

References

[1] 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.

[2] K. Iwama and E. Miyano, "An $O(\sqrt{N})$ oblivious routing algorithms for 2-D meshes of constant queue-size," In *Proc SODA'99* (1999) 466-475.

[3] C. Kaklamanis, D. Krizanc, S. Rao, "Simple path selection for optimal routing on processor arrays," In *Proc SPAA'92* (1992) 23-30.

[4] M. Kunde, "Routing and sorting on mesh connected processor arrays," In *Proc VLSI Algorithms and Architectures* (1988) 423-433.

[5] F.T. Leighton, "Average case analysis of greedy routing algorithms on arrays," In *Proc SPAA'90* (1990) 2-10.

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

[7] 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.

[8] R. Miller and Q.F. Stout, *Parallel algorithms for regular architectures: meshes and pyramids,* The MIT Press (1996).

[9] S. Rajasekaran and R. Overholt, "Constant queue routing on a mesh," *J. Parallel and Distributed Computing* 15 (1992) 160-166.

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

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