An Efficient Heuristic for Permutation Packet Routing on Meshes with Low Buffer Requirements

Fillia Makedon* Antonios Symvonis†

Abstract

Even though exact algorithms exist for permutation routing of n^2 messages on a $n \times n$ mesh of processors which require constant size queues, the constants are very large and the algorithms very complicated to implement. In this paper, we present a novel, simple heuristic for the above problem. It uses constant and very small size queues (size=2). For all the simulations we run on randomly generated data, the number of routing steps that is required by our algorithm is almost equal to the maximum distance a packet has to travel. We demonstrate a pathological case where the routing takes more than the optimal, and we prove that the upper bound on the number of required steps is $O(n^2)$. Furthermore, we show that our heuristic routes in optimal time inversion, transposition, and rotations, three special routing problems that appear very often in the design of Parallel Algorithms.

Index Terms- Mesh connected arrays, distance normalization, odd-even transposition, packet routing, permutations.

*Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH03755
†Basser Department of Computer Science, University of Sydney, Sydney, N.S.W. 2006, Australia

1 Introduction

An important task in the design of parallel computers is the development of efficient parallel data transfer algorithms [1] [9]. The two fundamental performance measurements here are, the number of parallel steps required to complete the message (packet) transfer and the additional buffer (queue) size needed for queueing in each processor. We study the packet routing problem, i.e., the problem of, given an interconnection network of processors, how to route the right data (packets), to the right processor fast. For this routing to be efficient, the processors must be able to communicate very fast with each other and the size of queues (buffers) created at any processor must be constant and very small. The requirement of small and constant buffer area for each processor is very important because it makes a parallel architecture scalable.

In this paper, we present a novel technique for packet routing on a mesh ($n \times n$ array) of processors. Our technique builts on the well known *odd-even transposition* method [2]. The main contribution of our technique is that it uses very small queues (buffer area for exactly 2 packets is needed). The algorithm is very simple and does not require complex operations for the maintenance of the buffers. So, it is a good candidate for hardware implementation. Furthermore, our experimental results show that the number of steps required to complete the routing is almost equal to the maximum distance a packet has to travel. Since the data for our experiments were obtained by random number generators, they suggest that the algorithm performs optimally with high probability.

We have chosen the mesh parallel architecture because its simple and regular interconnection pattern makes it especially suited for VLSI implementation. In addition, since the Euclidean distance between neighboring processing elements is constant on the mesh, the time needed for communication between any pair of connected elements is also constant. Furthermore, although the mesh has a large diameter ($2n - 2$ for an $n \times n$ mesh), its topology is well matched with many problems.

Previous work on packet routing includes deterministic [4] [5] [6] and probabilistic [3] [9]

approaches. The trivial greedy algorithm routes the packets to the correct column and then to the correct row in $2n-2$ steps. The size of the queues, however, can be as bad as $O(n)$. The nontrivial solutions given to this problem are based on parallel sorting algorithms [7] [8]. Kunde [4] [5] was the first to use parallel sorting to obtain a deterministic algorithm that completes the routing in $2n + O(n/f(n))$ steps with queues of size $O(f(n))$. Later on, Leighton, Makedon and Tollis [6] derived a deterministic algorithm that completes the routing in $2n-2$ steps using constant size queues.

The first probabilistic algorithm, derived by Valiant and Brebner [9], completed the routing in $3n + o(n)$ steps using $O(\log n)$ size queues. Later on, Krizanc, Rajasekaran, and Tsantilas [3] derived a probabilistic algorithm that routes the packets in $2n + O(\log n)$ steps using constant size queues. All probabilistic algorithms route the packets correctly with high probability. It is important to note here that, although two of the already known algorithms use constant-size queues, the size of the constant is too large and, as the authors admit, "*the constant bound in the resulting queuesize is not practical, even for moderate values of* n *(say* $n \leq 100$ *)*". Thus, efficient heuristics are needed, such as the one we propose, for practical applications in the design of parallel computers.

2 Preliminaries

A $n \times n$ *mesh of processors* is defined to be a graph $G = (V, E)$, where $V = \{(i,j) | 0 \leq i, j \leq n-1\}$, and an edge $e = ((i,j),(k,l))$ belongs to E if $|k-i| + |l-j| = 1$. The $n \times n$ mesh is illustrated in Figure 1. At any one step, each processor can communicate with all of its neighbors by the use of bidirectional links (called channels). We define the *distance* between two processors $P_1 = (i,j)$ and $P_2 = (k,l)$ as *distance*(P_1, P_2)=$|k-i| + |l-j|$. If *distance*(P_1, P_2)=1 then P_1 and P_2 are *neighbors*. It is convenient for the description of our algorithms to talk about the *east*, the *west*, the *north* and the *south* neighbors of a given processor. Formally, for processor $P = (i,j),\ 0 < i, j < n-1$, we define the east neighbor to be processor $P_e = (i, j+1)$, the west neighbor to be processor $P_w = (i, j-1)$, the north neighbor to be processor $P_n = (i-1, j)$, and

Figure 1: The 2-dimensional mesh.

the south neighbor to be processor $P_s = (i+1, j)$. A processors $P = (i, j)$ is said to be on the *boundary* of the mesh if $i = 0$ or $i = n-1$ or $j = 0$ or $j = n-1$. Processors on the boundary of the mesh do not have all of their four neighbors defined. All processors are assumed to work in a synchronous MIMD model.

In a *permutation problem*, each processor has one packet to transmit to another processor. At the end, each processor receives exactly one packet (1-1 routing). The problem here is to route all packets to their destinations fast and without the use of large additional storage area (buffers) in each processor.

Originally, the odd-even transposition method was used for sorting the elements of a linear array. This method works as follows: At odd time instances, the processors that are at odd array positions compare their contents with the contents of their east neighbors. A decision is made on whether an exchange of their contents will occur. At even time instances, a comparison and a possible exchange occurs between the processors at even positions of the array and their east neighbors. Figure 2 shows the comparisons that are performed at time $t = 0$ and $t = 1$ in an array of 8 elements.

Figure 2: Comparisons performed by the odd-even transposition method at time instances t=0 and t=1 for an array of 8 elements.

3 The Routing Algorithm

Our routing algorithm always tries to reduce the summation of the distances that all packets have to travel. Let $D(t)$ and $D(t+1)$ respectively be the total distance that all packets have to travel at time instances t and $t+1$. Then we will have that $D(t+1) \leq D(t)$. Furthermore, it is trivially true that, if the equality holds, then, at time instance $t+2$ we will have that $D(t+2) < D(t+1)$. This guarantees that eventually all packets will reach their destinations.

At time t, let processors P_i and P_{i+1} contain packets p_i and p_{i+1} and let the distance these packets have to travel be $d_i(t)$ and $d_{i+1}(t)$ respectively. We say that the distribution of the packets in processors P_i and P_{i+1} at time t is *normalized*, if the exchange of the contents of these processors with each other does not reduce the total distance $D(t)$ and also $\max(d_i(t), d_{i+1}(t)) \leq max(d_i(t+1), d_{i+1}(t+1))$, where $d_i(t+1), d_{i+1}(t+1)$ are the distances the packets have to travel after the exchange. An example of such a normalization of distances is shown at Figure 3. We call the packets whose contents are compared, *compared packets*.

Figure 3: Examples of Normalization

Algorithm *Route()* /* High level description. */

- Assume that processors in the same row of the mesh form a linear array in which odd-even transposition takes place.
- Perform odd-even transpositions on the rows of the mesh, as described in Section 2, such that the distances the compared packets have to travel are always normalized (packets are moving only horizontally).
- If a packet reaches the processor that lies at its column destination, and, no other packet at that processor wants to move vertically in the same direction (south or north), it starts its vertical movement (Figure 4).

  Otherwise, the packet that has to go further in the column, moves vertically, while, the other packet participates in the odd-even transposition that is taking place on the row (Figure 5).

Figure 4: Uninterrupted vertical movement of packets.

Figure 5: Interrupted vertical movement of packets.

Theorem 1 *Algorithm Route() needs a buffering area of exactly 2 packets.*

Proof Consider the situation where a queue can be created. Two packets enter a given node, the one is moving vertically, and, the other is moving horizontally but must change the direction of its movement because it has reached its column destination. If both want to move vertically in the same direction one must wait. Thus, a queue is created. In our routing algorithm this will not happen. Our algorithm has the property that, any two packets moving horizontally but in opposite directions cannot enter the same processor at the same time. This property comes from the application of the odd-even transposition. From the two packets that want to travel in the same vertical direction, the one that has to go further is chosen. The other, instead of being placed in a queue, participates in the odd-even transposition. So far, we have used buffer area of only one packet (the area for the packet that initially was in the processor). It remains to show how the communication between neighboring processors is implemented. Remember that, at any step, a processor communicates in the horizontal direction only with the neighbor on its east or on its west, but not with both. Consider any two neighboring processors that will communicate during the current step of the algorithm. Each one will transmit its contents to the other and, at the same time, it will keep a copy of its original packet. After the packets have been received, each processor computes the distance the two packets still have to travel before and after the exchange. If the distances are normalized after the exchange, then the previous contents of the packets are discarded. If the distances after the exchange are not normalized, then the exchange is ignored. Observe that both of the neighboring processors are doing exactly the same computations on the same data. Thus, they will reach compatible results. Also observe that by using the above described scheme, buffer area for only one extra packet is needed. This completes the proof. ∎

4 Performance

In this section, we present simulation results of the algorithm based on random routing problems on a 100×100 mesh connected array. We present a case where our algorithm takes more than

Table 1: Experimental results for random routing problems.

$2n$ steps, and, we prove that the algorithm completes the routing within $O(n^2)$ steps.

4.1 Experimental Results

We simulated our algorithm using random routing input data on a 100×100, a 50×50 and a 20×20 mesh connected array. The data were generated with the help of the random number generator function *drand48()* on a SUN workstation. Each experiment was set to start from a different point in the random sequence generated by this function. We run hundreds of experiments and we got near optimal performance on all of them (off by at most 1 routing step).

Table 1 shows results from some of our simulations on a 100×100 mesh connected array. The second column of this table contains the maximum distance that any packet has to travel in the specific experiment (This distance is always less than or equal to 198 since we are working on a 100×100 mesh). The fact that some entries in this column are the same does not mean that the data in the corresponding experiments are identical. The destinations of the 10000 packets in each experiment are totally different. The third column of Table 1 contains the number of the routing steps that are needed for the completion of the routing. Note that, this number is

Figure 6: A permutation of the packets that results in non-optimal routing time.

almost the same (off by at most 1) with the maximum distance in the corresponding experiment. This implies that the performance of the heuristic is near optimal.

4.2 A Low Performance Packet Routing Problem

In spite of the good experimental behavior of algorithm *Route()*, there exist permutations of the packets that result into routing time that is not optimal. In this section, we present such an initial setting of the packets. Assume that the destinations of the packets on the $n \times n$ mesh is as shown in Figure 6.

- All packets in the first row and in columns 0 trough $n - \sqrt{n} - 4$ are destined for the south-east $\sqrt{n} \times \sqrt{n}$ corner of the mesh.
- Packet P, initially at position $(0, n - \sqrt{n} - 3)$, is destined for the processor at position $(n - \sqrt{n} - 2, n - \sqrt{n} - 2)$.

Observe that, all packets that are initially located to the west of packet P have to travel further than P. So, during the odd-even transposition in the first row, packet P moves west since we always try to normalize the distances the packets have to travel. When it reaches the

Table 2: Experimental results on the number of delayed packets for random routing problems where non-optimal solutions are forced to occur.

processor at position (0,0) it starts moving to the east. It reaches column $n - \sqrt{n} - 2$, and then, moves vertically down until it arrives at its destination. The total movement of packet P takes $3(n - \sqrt{n})$ routing steps.

We have to mention that although there exists a case for which the required routing time is not optimal, no such case was produced by the random number generator. As is indicated in Table 2, the required number of steps for a problem for which the above "bad" situation is forced to occur, depends only on the distance the "bad" packet P has to travel. Furthermore, as it is indicated in Table 3, only a very small portion of the packets is not received in optimal time ("delayed" packets). More specifically, in all experiments only 38-40 packets out of an initial load of 10000 packets are "delayed" packets. When we traced these packets back to their origins, we found out that all of them originated in the first row of the mesh. The obvious conclusion from our experiments is that, the performance of the routing algorithm when such a "bad" situation occurs seems to depend only on the packets that constitute the "bad" situation. Again, we have to remind the reader that Tables 2 and 3 contain entries for a small portion of our simulations.

Table 3: Experimental results for random routing problems where non-optimal solutions are forced to occur.

4.3 An Upper Bound

Given the low performance routing problem described in the previous section, it is natural to ask for an upper bound on the number of routing steps required by algorithm *Route()* in order to solve any permutation routing problem. It is trivial to see that algorithm *Route()* cannot take more than $O(n^3)$ routing steps to route any permutation routing problem (At any routing step at least one packet will approach its destination, So the total distance that all packets have to travel is reduced by at least 1. The maximum total distance that all packets have to travel is $O(n^3)$. This implies that, after $O(n^3)$ routing steps all packets have reach their destinations). However, we will prove that algorithm *Route()* completes the routing within $O(n^2)$ steps.

Before we proceed with the proof we need the following definitions and lemmata. **Definition** A packet that at a given time instant participates in the odd-even transposition is said to be of type-H. **Definition** A row of the mesh is said to be empty at time t if at that time instant no packet of type-H is on the row.

Lemma 1 *Assume a row of the mesh with at most n packets of type-H. Also assume that no other packet will cross the row in order to reach its destination. Then, after* $2n$ *steps the row will be empty.*

Proof Observe that if no packet wants to cross the row from the north or the south, then the only packets that will participate in the odd-even transposition are the ones originally on that row. Another obvious observation is that it is impossible for two packets that were switched at a previous time instant to be compared and switched again. This implies that a packet can move at most n steps away of its destination. Then, after at most n steps, every packet will reach its column destination and will leave the row. ∎

Lemma 2 *Assume a row of the mesh with at most n packets of type-H. Also assume that at most k packets want to cross the row in order to reach their destination. Then, after* $2n + k$ *steps the row will be empty.*

Proof Similar to that of Lemma 1 ∎

Theorem 2 *Algorithm Route() will complete the routing of any permutation problem in* $O(n^2)$ *routing steps.*

Proof We will prove the theorem by showing that every row on the mesh will be empty after $O(n^2)$ steps. Then, after n steps all packets will reach their destinations since they are moving vertically and no collisions will happen. Lemma 1 implies that rows 0 and $n-1$ will be empty after $2n$ steps. We will use Lemma 2 to obtain a more general statement. Consider any row k, $1 \leq k < n-1$. At most $\min\{(k-1)n, (n-1-k)n\}$ packets want to cross row k in order to reach their destinations. From Lemma 2 we know that row k will be empty after $2n + \min\{(k-1)n, (n-1-k)n\}$ routing steps. The above expression gets its maximum value for $k = n/2$. This proves the theorem. ∎

The above theorem provides an upper bound on the number of routing steps required by

Algorithm *Route()*. However, we were not able to construct a routing problem that takes $O(n^2)$ steps for its completion by Algorithm *Route()*. We conjecture that Algorithm *Route()* can solve any permutation routing problem within $4n$ routing steps. If this is true, then, the analysis will be tight, since we have constructed a very complicated routing problem that needs about $4n$ steps for its completion. Also, we will have the first routing algorithm for the mesh that requires $O(n)$ routing steps and is not based on sorting of its submeshes. The existence of such an algorithm is an open problem.

4.4 An Interesting Property

In this section we prove a property of Algorithm *Route()* concerning the way the total distance that all packets have to travel is reduced during the course of the algorithm. Exploring similar properties might be the key tool in the effort to prove that a non-sorting based algorithm terminates after $O(n)$ steps.

Theorem 3 *Let $S_{i,t}$ denote the set of packets that are in row i at time t, and $D(S_{i,t}, l)$ denote the total distance that the packets in set $S_{i,t}$ still have to travel at time l, $l \geq 0$. Then, for every row i, $0 \leq i \leq n-1$, we have:*

$$D(S_{i,t}, t) \geq D(S_{i,t}, t+2) + 1 \quad \text{or}$$

$$D(S_{k,t}, t) \geq D(S_{k,t}, t+2) + 2 \quad \text{for} \quad k = i+1 \quad \text{or} \quad k = i-1.$$

Proof Consider all packets that are in row i at time t. They constitute set $S_{i,t}$. These packets can be divided into two categories. Packets that will participate in the odd-even transposition, already defined as type-H, and packets that will move vertically (to the north or to the south), we will call type-V. We consider two cases:

Case 1. $S_{i,t}$ contains no packet of type V.

First assume that there are two compared packets that want to travel to opposite directions. Then, they will be exchanged and we will have that $D(S_{i,t}, t) \geq D(S_{i,t}, t+1)+2$. At the next step these packets may continue their horizontal movement, start moving vertically, or do not move at all. In any case, the total distance is not decreased. Thus, $D(S_{i,t}, t) \geq D(S_{i,t}, t+2) + 2$.

Now assume that all compared packets want to move in the same direction. Even if an exchange takes place, the total distance that the packets have to travel remains the same. So, $D(S_{i,t}, t) = D(S_{i,t}, t+1)$. There are two possibilities. All the pairs of compared packets have the same direction, or there are two pairs that want to travel to opposite directions. If all the pairs have to travel to the same direction, then there must be less than n packets in the row. If there are n packets and, say, they are all moving to the left, the leftmost one must be in its correct column, so it must be of type V. But we assumed that no such packet exists. So, there are less than n packets in the row. This implies that there exist a "gap", and, a packet must be adjacent to it. During the next step, the packet that is adjacent to the "gap" will occupy it. Thus, we get that $D(S_{i,t}, t) \geq D(S_{i,t}, t+2) + 1$. The second possible case is that there are two pairs of packets that want to move to opposite directions. Then, during the next step, their adjacent packets will be compared and an exchange will occur. Thus, we will have that $D(S_{i,t}, t) \geq D(S_{i,t}, t+2) + 2$. If there are no such adjacent pairs, there must be a "gap". In this case we will have that $D(S_{i,t}, t) \geq D(S_{i,t}, t+2) + 1$.

Case 2. There is at least one packet of type V.

Let P be such a packet. Then P moves vertically and at time $t+1$ we have: $D(S_{i,t}, t) \geq D(S_{i,t}, t+1) + 1$. Now, we have to consider what will happen in the next step. If packet P stays at its correct column during the next step, then, we have that $D(S_{i,t}, t) \geq D(S_{i,t}, t+2) + 1$. Assume now, that, in the next step it is forced to move away from its destination. Then, we might have that $D(S_{i,t}, t) = D(S_{i,t}, t+2)$. Packet P is now at row k, where k is $i+1$ or $i-1$. We will show that $D(S_{k,t}, t) \geq D(S_{k,t}, t+2) + 2$. The fact that packet P is forced to move away from its destination implies two things. First of all, there is another

packet at row k which is destined for the same column that packet P is, and, in addition, it has to go further. This packet forces P to participate in the odd-even transposition. Secondly, there is another packet that forces P to move out of its column destination. Thus, there are two packets in set $S_{k,t}$ that each reduces its distance by one step. So, we have that $D(S_{k,t}, t) \geq D(S_{k,t}, t+2) + 2$.

∎

5 Routing Problems that Can Be Solved Optimally

In this section, we consider some special forms of permutation routing problems for which algorithm *Route()* performs optimally. We prove that when the permutation problem is a *rotation*, an *inversion*, or a *transposition*, then at most $2n$ steps are required. Before we procced to examine these problems, we prove two useful lemmas that concern permutation routing on a chain of processors.

5.1 Permutation Routing on a Chain of Processors

Lemma 3 *A permutation routing problem on a chain of n processors can be solved in n steps using the odd-even transposition method.*

Proof A way to solve the permutation routing problem is by sorting all packets according to their destination addresses. Since the odd-even transposition method is used for sorting in linear arrays, the time required for sorting is also enough for routing. But, we know that in order to sort n elements on a linear array of size n, we need exactly n steps if the odd-even transposition method is used [2]. ∎

Lemma 4 *A permutation routing problem on a chain of processors can be solved in r routing steps, where r is the maximum distance a packet has to travel.*

Proof We can achieve the number of steps that is stated at the lemma if all packets start their motion at the same time, and, always travel towards their destinations. Then, nothing can delay a packet. Since at each step every packet approaches its destination, the packet that has to travel distance r will do so in r routing steps. ∎

5.2 Rotations

Definition: *A permutation packet routing problem on a $n \times n$ mesh is called an (l, m)-rotation if the packet initially at position (i, j) is destined for position $(i + l \bmod n, j + m \bmod n)$, where $0 \leq i, j \leq n - 1$ and $0 \leq l, m \leq n - 1$.*

If $l = 0$ we have a horizontal rotation, if $m = 0$ a vertical rotation, and if $m = l$ a diagonal rotation. The next theorem shows that for all kinds of rotations algorithm *Route()* is optimal.

Theorem 4 *Given an $n \times n$ mesh of processors, algorithm Route() needs at most $n + \max(l, n - l - 1)$ steps in order to complete the routing of any permutation problem that is an (l, m)-rotation, for $0 \leq l, m \leq n - 1$.*

Proof For the moment, assume that algorithm *Route()* routes all packets in two consecutive and time disjoint phases. During the first phase, it routes all packets horizontally until they reach their column destination. The second routing phase starts when all the packets reach their column destination. All packets are routed to their final destination along the columns of the array. Observe now, that when the second phase starts, each processor at any column has exactly one packet to route vertically. This is so, because in the beginning of the routing all the packets of the same row are destined for different columns. Thus, in each phase of the algorithm, we have a permutation problem to solve. Since we use the odd-even transposition method in order to route the packets during the first phase, we know from Lemma 3 that n steps are enough. During the second phase the routing is done as described in Lemma 4. So, we need exactly $\max(l, n - l - 1)$ more steps in order to complete the routing. Thus, a total of $n + \max(l, n - l - 1)$ steps is required.

In the above analysis, we assumed that the packets are routed in two time disjoint phases, an assumption that is not true. However, the analysis is still valid. To see why, observe that, all packets that are destined for the same column will reach that column at the same step. Thus, all packets that are destined for the same column start their column routing at the same time. This synchonization permits us to treat the movement of the packets as two separate phases, a horizontal, and a vertical one. This completes the proof. ∎

5.3 Inversion

Definition: *A permutation packet routing problem on a $n \times n$ mesh is called an inversion if the packet initially at position (i, j) is destined for position $(n-1-i, n-1-j)$, where $0 \leq i, j \leq n-1$.* Theorem 5 shows that algorithm *Route()* performs optimally if the routing problem is an inversion.

Theorem 5 *Given an $n \times n$ mesh of processors, algorithm Route() needs exactly $2n-1$ steps in order to complete the routing of an inversion packet routing problem.*

Proof All packets that are destined for column $n-1-j$, $0 \leq j \leq n-1$, are initially located at column j. Since the movement of the packets in all columns is similar, all the packets that are destined for the same column reach that column at the same step. So, as in the proof of Theorem 4, we can assume in our analysis that the routing is done at two time disjoint phases. From Lemma 3, we know that the first phase needs exactly n routing steps. The maximum distance a packet has to travel during the second phase is $n-1$. This is because, all packets that initially are located in the first row of the mesh have destinations in the last row of the mesh. By applying Lemma 4, we conclude that the second phase takes exactly $n-1$ steps. Thus, algorithm *Route()* solves the inversion packet routing problem in exactly $2n-1$ steps. ∎

5.4 Transposition

Definition: *A permutation packet routing problem on an $n \times n$ mesh is called a transposition if the packet that initially is at position (i, j) is destined for position (j, i), where $0 \leq i, j \leq n-1$.*

Theorem 6 *Given an $n \times n$ mesh of processors, algorithm Route() needs exactly $2n-2$ routing steps in order to solve the transposition routing problem.*

Proof Without loss of generality consider packet P that initially is located at position (i, j), where $i \leq j$. The case where $i > j$ is treated similarly. Packet P is destined for location (j, i). Observe that, all the packets to its left are also destined for column i, but, they have to travel greater distance. More specifically, the packet that is k positions to its left has to travel $2k$ step more than P. So, P is "trapped" by the packets to its left. As a result, it moves to the left until it hits the left boundary of the mesh. Then it moves uninterrupted to the right until it reaches its column destination. Finally, it moves vertically to its destination. Figure 7 illustrates the motion of packet P. Note that no interaction of packets occurs during the vertical routing because the packets that are to the left of column i want to move upwards, while those to the right want to move downwards. The travel of packet P, initially at position (i, j) takes a total of $j + i + (i - j) = 2i$ routing steps. This implies that all packets reach their destinations after $2n - 2$ steps, since the maximum value that i can take is $n - 1$. ∎

6 Conclusion and Further Work

We have presented an efficient heuristic for routing messages (packets) on mesh connected parallel computers. The main advantage of our algorithm is that it uses buffer area of exactly 2 packets per processor.Furthermore, the algorithm is simple and its experimental behavior indicates that it performs optimally for random routing problems. It is the first time, according to our knowledge, that such a heuristic is reported, and its surprisingly good experimental results make it more important. An interesting problem would be to prove that this algorithm works

Figure 7: The motion of packet P during the routing of a transposition.

optimally with high probability. However, the fact that the movement of any packet depends on the destinations of its neighboring packets at any time instance makes this problem a very difficult and challenging one. Another interesting problem is to provide a tighter analysis of Algorithm *Route()*. There is a gap between the upper bound of $O(n^2)$ routing steps and the $4n$ performance of the worst case routing problem we were able to construct.

References

[1] W.J. Dally, C.L. Seitz, "The Torus Routing Chip", Distributed Computing (1), pp. 187-196, 1986.

[2] D.E. Knuth, The Art of Computer Programming, volume 3, pp 241, Addison Wesley, 1972

[3] D. Krizanc, S. Rajasekaran, Th. Tsantilas, "Optimal Routing Algorithms for Mesh-Connected Processor Arrays", VLSI Algorithms and Architectures (AWOC'88), J. Reif, ed, Lecture Notes in Computer Science 319, 1988, pp. 411-422.

[4] M. Kunde, "Routing and Sorting on Mesh-Connected Arrays", VLSI Algorithms and Architectures (AWOC'88), J. Reif, ed, Lecture Notes in Computer Science 319, 1988, pp. 423-433.

[5] M. Kunde, "Packet Routing on Grids of Processors", unpublished manuscript.

[6] F.T. Leighton, F. Makedon, I.G. Tollis, "A $2n-2$ Algorithm for Routing in an $n \times n$ Array With Constant Size Queues", Proceedings of ACM Symposium on Parallel Algorithms and Architectures, SPAA'89, June 1989, pp. 328-335.

[7] C.P. Schnorr, A. Shamir, "An Optimal Sorting Algorithm for Mesh Connected Computers", In proc. 18th ACM Symposium on Theory of Computing, 1986, pp.255-263.

[8] C.D. Thompson, H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer", Comm. ACM, 20 (1977) pp. 263-270.

[9] L.G. Valiant, "A Scheme for Fast Parallel Communication", SIAM J. Comp. 11 (1982), pp. 350-361.