Routing on Trees via Matchings*

ALAN ROBERTS[1], ANTONIS SYMVONIS[1] and LOUXIN ZHANG[2]

[1] Department of Computer Science, University of Sydney, N.S.W. 2006, Australia
[2] Department of Computer Science, University of Waterloo, Canada, N2L 3G1

Abstract. In this paper we consider the routing number of trees, denoted by $rt()$, with respect to the matching routing model. The only known result is that $rt(T) \leq 3n$ for an arbitrary tree T of n nodes [2, 3]. By providing off-line permutation routing algorithms we prove that: i) $rt(T) \leq n + o(n)$ for a complete d-ary tree T of n nodes, ii) $rt(T) \leq 2n + o(n)$ for an arbitrary bounded degree tree T of n nodes, iii) $rt(T) \leq 2n$ for a maximum degree 3 tree T of n nodes, iv) $rt(T) \leq \frac{13}{5}n$ for an arbitrary tree T of n nodes.

1 Introduction

The *permutation packet routing problem* on a connected undirected graph is the following: We are given a graph $G = (V, E)$ and a permutation π of the vertices of G. Every vertex v of G contains a packet destined for $\pi(v)$. Our task is to route all packets to their destinations.

During the routing, the movement of the packets follows a set of rules. These rules specify the *routing model*. Let $rt_M(G, \pi)$ be the number of steps required to route permutation π on graph G by using routing model M. The routing number of graph G with respect to routing model M, $rt_M(G)$, is defined to be $rt_M(G) = \max_\pi rt_M(G, \pi)$ over all permutations π of the vertex set V of G.

The routing number of a graph was first defined by Alon, Chung and Graham in [2, 3]. In their routing model, the only operation allowed during the routing is the exchange of the packets at the endpoints of an edge of graph G. The exchange of the packets at the endpoints of a set of disjoint edges (a matching on G) can occur in one routing step. We refer to this model as the *matching routing model* and, for a graph G, we refer to the routing number of G with respect to the matching routing model, simply as the routing number of G, denoted by $rt(G)$. It was shown in [2, 3] that $rt(T) < 3n$ for any tree T of n vertices. As a consequence, $rt(G) < 3n$ for any graph G of n vertices. To the best of our knowledge, this is the only known work on routing on trees under the matching model. Algorithms for routing permutations on trees under different routing models have been presented by Borodin, Rabani and Schieber [4] (hot-potato routing model) and Symvonis [9] (simplified routing model).

In our attempt to obtain an upper bound on the routing number of complete d-ary trees, we run into a problem of independent interest. This is the problem of *heap construction*. Consider a rooted tree T and let each of its nodes have a *key-value* associated with it. We say that T is *heap ordered* if each non-leaf node satisfies the *heap invariant*: "the key-value of the node is not larger than the key-values of its

* The work of Dr Symvonis is supported by an ARC Institutional Grant.
Email: {alanr,symvonis}@cs.su.oz.au, lzhang@neumann.uwaterloo.ca.

children". When the key-value at each node is carried (or associated with) the packet currently in the node, the problem of heap construction is simply to route the packets on the tree in a way that guarantees that at the end of the routing the packets are heap-ordered based on the key-values they carry. Needless to say, we are interested in forming the heap in the smallest number of parallel routing steps when routing is performed according to the matching routing model. Heaps are also discussed in the context of the PRAM model [5, 7, 10]. Rao and W. Zhang [7] and W. Zhang and Korf [10] described how to construct a heap (implemented as a complete binary tree) of n elements within $2\log_2 n$ steps.

1.1 Our Results

In this paper, we consider the routing number of several classes of trees with respect to the matching routing model. We present an off-line algorithm which routes any permutation on a complete d-ary tree within $n + o(n)$ steps. Firstly we describe how to route a permutation on an n-node complete binary tree in $\frac{4}{3}n + o(n)$ routing steps. Then, we extend the algorithm to route a permutation on an n-node complete d-ary tree in $(1 + \frac{1}{d^2-1})n + o(n)$, and finally we extend the later algorithm to achieve a routing time of $n + o(n)$ steps.

We also present an algorithm that routes a permutation on a bounded-degree tree of n nodes within $2n + o(n)$ routing steps. For trees of maximum-degree 3, the algorithm requires at most $2n$ routing steps. Our algorithm can be considered to be an extension of the algorithm of Alon, Chung and Graham [2, 3]. We manage to show that it is possible to partially overlap the first two major steps of their algorithm. For arbitrary trees of n nodes, application of this idea results in an algorithm that routes any permutation within $\frac{13}{5}n$ steps.

During the course of our off-line tree routing algorithms, we need to have the tree heap-ordered with respect to key-values assigned to the packets at its nodes. For this reason, we use an algorithm that can be considered to be an extension of the odd-even transposition method. The same algorithm was used for building a heap priority queue (i.e., a complete binary tree) in an EREW-PRAM environment [7, 10]. We show that an arbitrary rooted tree of height h can be heap-ordered within $2h$ routing steps. While the proof originally reported in [7] appears to generalise to arbitrary trees, we provide an analysis based on potential functions.

The paper is organised as follows: In Section 2, we present definitions used throughout the paper. In Section 3, we present the heap construction algorithm. Sections 4 and 5 are devoted to routing on complete trees. In Section 6, we consider routing on bounded degree trees and arbitrary trees. Space limitations force us to omit a large amount of technical details, including almost all proofs. These details can be found in [8].

2 Preliminaries

A *tree* $T = (V, E)$ is an undirected acyclic graph with node set V and edge set E. Throughout the paper we use standard graph theoretic terminology and we assume n-node trees, i.e., $|V| = n$. The *depth* $d_T(v)$ of node v is defined to be the distance

from the root r to v. The *height* of tree T, denoted by $h(T)$, is defined to be $h(T) = max_{v \in V(T)} d_T(v)$. We say that node v is a *level-i node* (or, at *depth-level i*) if $d_T(v) = i$. The root of the tree is a level-0 node. We say that edge e is a *level-i edge* if it connects a level-i node with a level-$(i+1)$ node. All edges connected to the root r are level-0 edges. We denote by $lca(v, u)$ the *lowest common ancestor* of nodes v and u.

To comply with the usual drawing of rooted trees in which the root of a subtree is the topmost node of its drawing, we give some additional definitions. We say that a level-i node u is *below* a level-j node v if v is an ancestor of u (it is also implied that $i \leq j + 1$). Equivalently, we say that node v is *above* node u. When v is the parent of u we use the terms *immediately above* and *immediately below*. Similar terminology is used for edges with respect to other edges/nodes.

A *subtree rooted at v*, denoted by T_v, consists of v, all descendants of v and the edges between them. A *partial tree* is a connected subgraph of a tree. (Note the difference between a subtree and a partial tree.) By T^i we denote the partial tree of T which is rooted at r and contains all level-j nodes, $0 \leq j \leq i$.

A *d-ary tree*, $d \geq 2$ is defined to be a rooted tree of which all internal nodes have exactly d children. A d-ary tree T is said to be *complete* if all of its leaves are level-$h(T)$ nodes. A complete d-ary tree has exactly d^i level-i nodes and its height is $h(T) = \lfloor \log_d n \rfloor$, where n is the number of nodes of the tree. We use a special naming convention for the nodes of complete d-ary tree T; the root r of the tree is denoted by $r_{(0,1)}$, and the children of the internal node $r_{(i,j)}$, $0 \leq i < h(T)$, $1 \leq j \leq d^i$ are $\{r_{(i+1,k)} \mid d(j-1)+1 \leq k \leq dj\}$. When we draw a complete d-ary tree we position node $r_{(i,j)}$ above node $r_{(k,l)}$ if $i < k$. We position node $r_{(i,j)}$ to the left of node $r_{(i,k)}$ if $j < k$. Figure 1 shows a complete ternary tree using the introduced naming and layout conventions.

Most of the work available on the routing number of trees is based on off-line recursive routing algorithms. In order to be able to apply recursion, we must identify subtrees in which all packets destined for nodes each subtree have arrived in it. We say that, for a given packet, a subtree is a *destination subtree* if it contains the node the packet is destined for. The following lemma considers the situation where we want to route packets to their destination subtree.

Lemma 1. *Consider a tree T (rooted at r) and a permutation to be routed. Let m be the number of packets that have to cross the root r in order to reach their destination and assume that these packets form partial trees rooted at the children of r. Then these m packets can be routed into their destination subtrees (rooted at children of r) within at most $m + d - 1$ steps, where d is the degree of r.*

3 Heap Construction on Rooted Trees

Consider a rooted tree T and let each node of T initially contain a packet. Moreover, let each packet have a *key-value* associated with it, with all key-values drawn from a totally ordered set. Our objective is to route the packets in such a way that, at the end of the routing, the tree is heap-ordered with respect to the key-values at its nodes.

We describe an algorithm that completes the task within $2h(T)$ routing steps. The algorithm works for arbitrary rooted trees and is a generalisation of the odd-even transposition sorting method [6, 1].

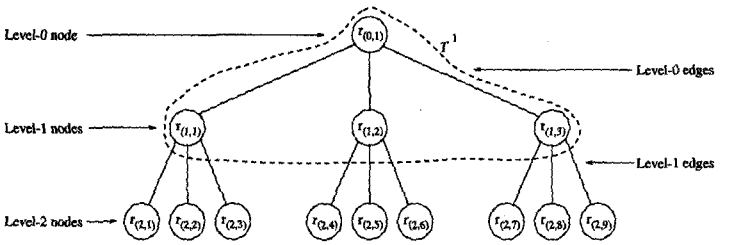

Fig. 1. A complete ternary tree.

Algorithm *Odd-Even_Heap_Construction*(T)
/* W.l.o.g., we assume that all key-values associated with the packets are distinct */

1. Assign label $h(T) - (i+1)$ to each level-i edge, $0 \leq i < h(T)$.
2. $t = 0$
3. **While** $t \leq 2h(T)$ **do**
   (a) For any node u with edges connecting to its children labelled congruent to t mod 2, select out of the children of u the child, say v, that contains the packet of the smallest key-value. Order for a comparison between the key-values of the packets at u and v to take place at time t. If the key-value of the packet at v is smaller than the key value of the packet at u, a swap of the packets takes place.
   (b) $t = t + 1$

See [8] for a proof (based on a potential function) of the following theorem:

Theorem 2. *Algorithm* Odd-Even_Heap_Construction() *heap-orders any tree T in at most $2h(T)$ steps, where $h(T)$ is the height of the tree T.*

4 Routing on Complete Binary Trees

Consider the complete binary tree T of n nodes (Figure 2). Recall that a subtree rooted at node x is denoted by T_x and that by T^i we denote the complete partial tree of T of depth i.

Algorithm *Route_on_Complete_Binary_Trees* /* *RCBT*, for short. */

1. Assign to every packet destined for a node in T^1 a class-value of 0. The remaining packets are assigned a class-value of 1.
2. Heap-order T with respect to the class-values of its packets.

3. Route the class-0 packets into the subtrees rooted at level-2 nodes such that each such subtree contains at most 1 class-0 packet.

4. Partition the packets into classes. A class-value is assigned to every packet based on its current position and its destination as follows:

Current node	Destination	Class-value	
$T_{r_{(1,1)}}$	$T_{r_{(2,3)}} \cup T_{r_{(2,4)}}$	0	
$T_{r_{(1,2)}}$	$T_{r_{(2,1)}} \cup T_{r_{(2,2)}}$	0	
r	$T_{r_{(2,1)}} \cup T_{r_{(2,2)}} \cup T_{r_{(2,3)}} \cup T_{r_{(2,4)}}$	0	
$T_{r_{(2,1)}}$	$T_{r_{(2,2)}}$	1	
$T_{r_{(2,2)}}$	$T_{r_{(2,1)}}$	1	
$T_{r_{(2,3)}}$	$T_{r_{(2,4)}}$	1	
$T_{r_{(2,4)}}$	$T_{r_{(2,3)}}$	1	
T	$\{r, r_{(1,1)}, r_{(1,2)}\}$	2	
$T_{r_{(2,i)}}$	$T_{r_{(2,i)}}$	3	$i = 1, 2, 3, 4$
$r_{(1,1)}$	$T_{r_{(2,1)}} \cup T_{r_{(2,2)}}$	2^*	
$r_{(1,2)}$	$T_{r_{(2,3)}} \cup T_{r_{(2,4)}}$	2^*	

5. Heap-order tree T with respect to the class-values of their packets. During the heap construction, update the class-values of packets as follows:

   - When a class-1 packet reaches the root of T, the packet immediately becomes a class-0 packet.
   - When a class-2^* packet enters its destination subtree, it immediately becomes a class-3 packet.
   - When a class-2^* packet enters a subtree that does not contain its destination, it immediately becomes a class-1 packet.
   - If at the end of the heap construction a class-2^* packet is still at a level-1 node, it becomes a class-1 packet.

6. Route the packets to their destination subtrees (T^1, and $T_{r_{(2,i)}}$, $i = 1, 2, 3, 4$).

7. Recursively route the packets in T^1, and $T_{r_{(2,i)}}$, $i = 1, 2, 3, 4$.

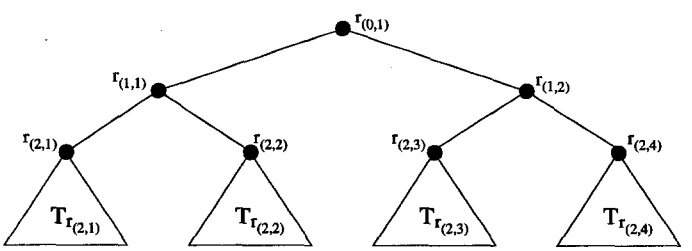

Fig. 2. A complete binary tree.

The first 5 steps are executed in order to ensure that the positions of the packets after Step 5 allow for the mplementation of Step 6 within at most $n + 2$ steps.

Some explanations are necessary regarding Steps 4 and 5 and the class updates that happen for some packets. According to the algorithm, if during the heap construction a class-1 packet reaches the root of T, then that packet becomes a class-0 packet. The reason for doing that is to stop packets from crossing the root to the other side of the tree. The class update does not affect the analysis of the heap construction algorithm since the packet will remain at the root till the end of the heap construction. Another possible class update is that of class-2* packets which become either class-1 or class-3 packets. Recall that a class-2* packet is initially located at a level-1 node and is destined for one of the subtrees rooted at its children. Firstly observe that during the heap-construction at Step 5, it is not possible for a class-2* packet to move to the root of T. This is because it will require that the class-2* packet is swapped with a class-3 packet which was at the root, a situation which cannot occur. Thus, the class-2* packets will either move to a node towards the leaves of the tree or they will remain in their current positions. In the case where a class-2* packet enters its destination subtree, we update it to a class-3 packet. Note that this update will not cause any problems to the heap construction algorithm, because the packet never moved upwards in the tree. If the class-2* packet enters the subtree which does not contain its destination, it becomes a class-1 packet. Again this class change does not cause any problems to the heap construction algorithm.

If during the heap construction a class-2* packet does not move, i.e., it remains at a level-1 node, it is updated to a class-1 packet. This update happens at the end of the heap construction algorithm and certainly does not affect its complexity.

A final comment regarding the first three steps of the algorithm. The purpose of these steps is to distribute the class-2 packets in a balanced way among subtrees $T_{r(1,1)}$ and $T_{r(1,2)}$. These steps are not really necessary. If we omit them we can still perform the routing on a complete binary tree in $\frac{4}{3}n + o(n)$ steps with the algorithm of this section but, the refinement of Step 6 that follows would be more complicated.

4.1 Refinement of Step 6

During the routing of Step 6, packets that enter their correct subtree continue to move towards the bottom of the subtree. Their movements stops when they reach a node such that all descendent nodes of it hold class-3 packets. This movement of packets towards the bottom of their destination subtree guarantees that the packets that have to leave the subtree will always be close to the root of the subtree.

Also note that no problems occur during the movement of a packet towards the leaves of the tree if the packet is swapped with another packet that wants to move upwards. This is because both packets move towards their destination subtree and that movement can be easily accommodated. The problems start when a packet that moved upwards because of a swap with another packet, reaches the node (other than the root) where it has to switch direction and start moving towards the leaves of the tree. In Step 6, this can only happen at the 2 level-1 nodes. This is because we are only interested in routing the packets to their destination subtree, not to their destination node. The reason that we would like the packets to switch the direction of their movement is because we do not want them to cross from one side of the tree to

the other. However, this change of direction requires that for consecutive steps level-1 edges adjacent to the same node are active. This has an effect on the flow of packets from $T_{r_{(1,1)}}$ to $T_{r_{(1,2)}}$ and vice versa.

In order to bound the number of routing steps required by Step 6 of the algorithm, we assume for the moment that the flow of packets is uninterrupted. This means that packets that are moving towards the leaves of the tree proceed uninterrupted and, as a result of their movement, some packets that have to move upwards during the trip to their destination subtrees do so (later on, these packets will reach the root and they will start their movement towards the leaves). What we described above will actually happen if we have to route a permutation in which all packets initially in $T_{r_{(1,1)}}$ are destined for $T_{r_{(1,2)}}$ and vice versa. In this case the analysis of the routing is quite easy. However, we have to route permutations that do not conform with the above pattern. In these permutations, packets have to change direction at level-1 nodes.

In order to deal with the problem of packets that want to turn at level-1 nodes we are going to use (in the description of the refinement of Step 6) the *"freeze command"* which freezes any routing that takes place at parts of the tree. During these *"frozen steps"* packets are able to change direction at a level-1 node. The time spent on these "freeze commands" is time well spent. Assume that we have an initial estimate on how difficult (in terms of required routing steps) the routing problem at hand is. As we will see, whenever we freeze the routing for one step, we will be able to update our estimate regarding the required number of routing steps. More specifically, for each freeze command we execute, we can save at least 1 routing step from our initial estimate.

After Step 5, the class 0 packets form a partial subtree rooted at $r_{(0,1)}$. The routing of Step 6 will mainly concentrate on packets of class 0 and will move them to their correct subtree $T_{r_{(2,i)}}$, $i = 1, 2, 3, 4$. We now describe how this is done. W.l.o.g., assume that the packet p at $r_{(0,1)}$ is destined for $T_{r_{(2,1)}}$. All other cases can be handled symmetrically. After the first swap, p will find itself at node $r_{(1,1)}$. Now, we have four cases to consider:

1. $r_{(2,1)}$ *contains a class-0 packet.* Let q be the packet at node $r_{(2,1)}$. In this case, p will swap with q and it will enter its destination subtree $T_{r_{(2,1)}}$. At this stage, it "becomes" a class-3 packet. (In the following steps, it will continue moving towards the bottom (leaves) of the tree until it reaches a node of which all descendants are class-3 packets.) If at the same time that the swap of p and q was taking place, $r_{(0,1)}$ gets a new class-0 packet, q will be swapped with it in a subsequent step and thus, the flow of class-0 packets continues.

2. $r_{(2,1)}$ *contains a class-1 packet.* Let q be the packet at node $r_{(2,1)}$. We distinguish the following cases based on the class of the packet, say q', at node $r_{(2,2)}$.

   (a) $r_{(2,2)}$ *contains a class-0 packet.* We swap p with q and then we issue 1 *freeze* command. During this extra step, q (now at $r_{(1,1)}$) will swap with q'. After this swap, q' will be able to move towards its destination subtree (through a subsequent swap with the packet at $r_{(0,1)}$) and q reaches its destination subtree $T_{r_{(2,2)}}$ and starts moving towards the leaves.

   (b) $r_{(2,2)}$ *contains a class-1 packet.* This is an interesting case. Firstly observe that the class-0 packets in $T_{r_{(1,1)}}$ are exhausted. We swap p with q and then we issue 1 *freeze* command. During this extra step, q (now at $r_{(1,1)}$)

will swap with q' and p (now at $r_{(2,1)}$) will swap with the packet of smaller class immediately below it (one must exist). Now, after the first *freeze* command, we have at node $r_{(1,1)}$ a class-1 packet and at node $r_{(2,1)}$ either a class-1 or a class-2 packet. In the case where there is a class-2 packet at node $r_{(2,1)}$, we issue a final *freeze* command and during the extra step we swap the packets at nodes $r_{(1,1)}$ and $r_{(2,1)}$. In the case where there is a class-1 packet at node $r_{(2,1)}$ we also execute a *freeze* command and make the same swap but this *freeze* command is not the last one. More specifically, we continue issuing *freeze* commands and making swaps that have as a result to route class-1 packets to their destination subtree. This sequence of *freeze* commands ends when a class-2 packet arrives at node $r_{(1,1)}$. Such a class-2 packet must exist since we assumed that the packet p that triggered the sequence was a class-0 packet.

(c) $r_{(2,2)}$ *contains a class-2 packet.* We swap p with q and then we execute a *freeze* command. During the extra step, q (now at $r_{(1,1)}$) will swap with q'. This case can be considered as a special case of 2b where the sequence of class-1 packets consists of only 1 packet.

(d) $r_{(2,2)}$ *contains a class-3 packet.* This is impossible (because of the initial sorting and the fact that we are routing a permutation).

3. $r_{(2,1)}$ *contains a class-2 packet.* Let q be the packet at node $r_{(2,1)}$. We distinguish the following cases based on the class of the packet, say q', at node $r_{(2,2)}$.

   (a) $r_{(2,2)}$ *contains a class-0 packet.* Swap p with q and then issue a *freeze* command. During the extra step, swap q with q'. This will keep q from being trapped into $T_{r_{(1,2)}}$ (there might be many class-0 packets that want to reach $T_{r_{(1,1)}}$). In subsequent steps q will move towards the bottom of $T_{r_{(2,2)}}$ but it will always stay above class-2 or class-3 packets in any leaf-to-root path.

   (b) $r_{(2,2)}$ *contains a class-1 packet.* We simply swap p with q. Note that because of the initial balancing of the class-2 packets, there is no chance that q will be trapped into $T_{r_{(1,2)}}$. No *freeze* command is needed.

   (c) $r_{(2,2)}$ *contains a class-2 packet.* As in Case 3b.

   (d) $r_{(2,2)}$ *contains a class-3 packet.* As in Case 3b.

4. $r_{(2,1)}$ *contains a class-3 packet.* This is impossible. The fact that the packets were heap-ordered together with the fact that $r_{(2,1)}$ contains a class-3 packet imply that all packets in $T_{r_{(2,1)}}$ have destinations inside it. Then, p must have a destination outside $T_{r_{(2,1)}}$

Note that when the routing of class-0 packets finishes, we must have at node $r_{(1,1)}$ a class-2 packet. Thus, all packets at the subtrees below it must contain only class-3 packets which in turn implies that the routing of Step 6 is over (for $T_{r_{(1,1)}}$).

Lemma 3. *The routing that occurs during Step 6 of Algorithm RCBT terminates after at most $n + 2$ routing steps.*

Theorem 4. *Algorithm RCBT routes in an off-line fashion a permutation on a complete binary tree of n nodes in at most $\frac{4}{3}n + o(n)$ steps.*

5 Routing on Complete d-ary Trees

The algorithm is a generalisation of algorithm *Route_on_Complete_Binary_Trees*.

Algorithm *Route_on_Complete_d-ary_Trees* /* *RCdT*, for short. */

1. Identical to Step 1 of Algorithm *RCBT*.
2. Identical to Step 2 of Algorithm *RCBT*.
3. Identical to Step 3 of Algorithm *RCBT*.
4. Partition the packets into classes. A class-value is assigned to packet p, currently at node *curr* and destined for node *dest*, as follows:
   - Let l be the level of the lowest common ancestor of nodes *curr* and *dest*, i.e., $l = d_T(lca(curr, dest))$.
   - **If** *dest* is in T^1 **then** p is a class-2 packet
     else if *curr* is a level-1 node and *dest* is in T_{curr} **then** p is a class-2* packet
     else if $l > 1$ **then** p is a class-3 packet
     else p is a class-l packet[3].
5. Identical to Step 5 of Algorithm *RCBT*.
6. Route the packets to their destination subtrees (T^1, and $T_{r_{(2,i)}}$, $1 \leq i \leq d^2$).
7. Recursively route the packets in T^1, and $T_{r_{(2,i)}}$, $1 \leq i \leq d^2$.

Lemma 5. *The routing that occurs during Step 6 of Algorithm RCdT terminates after at most* $n + d^2 - 1$ *routing steps.*

Proof. To prove the lemma we have to describe the details of the routing in Step 6. We can distinguish cases as we did for algorithm *RCBT* but this will be a tedious repetition. Thus, we only describe what are the differences between the routing that takes place at Step 6 of algorithm *RBCT* and that of algorithm *RBdT*.

Since in algorithm *RCBT* we concerned with binary trees, in all sub-cases of parts 2 and 3 of the refinement of Step 6 there was only one sibling of node $T_{r_{2,1}}$ to consider. When routing in complete d-ary trees with algorithm *RCdT*, all sub-cases must be updated to account for the fact that each level-2 node has exactly $d - 1$ siblings. So, in the refinement of Step 6 of algorithm *RCdT* case 2(a) is changed to: *"RBdT* : 6.2(a) *There exists a sibling node of* $T_{r_{2,1}}$ *which contains a class-0 packet"*, case 2(b) is changed to: *"RBdT* : 6.2(b) *There exists a sibling node of* $T_{r_{2,1}}$ *which contains a class-1 packet"*, and so on. Most importantly, these sub-cases are organised in an **"if ... then ... else if ... else ..."** statement. We only proceed to case 2(b) if we fail to locate a node satisfying case 2(a).

As in the analysis of algorithm *RCBT* (Lemma 3, [8]), we can account for all "frozen steps" which involve class-1 packets. However, we have to pay the extra cost for the cases that a class-2 packet (located at a level-1 node) is swapped with a class-0

[3] Note that for complete binary trees the class assigned to each packet is identical with the class assigned by algorithm *Route_on_Complete_Binary_Trees*.

packet (located at a level-2 node). In this case, no class-2 packet will enter the same subtree (rooted at a level-2 node) twice. Because of the initial balancing, at most 2 class-2 packets are in each subtree rooted at a level-1 node. This implies that the extra number of routing steps for a given subtree rooted at a level-1 node is at most $2(d-1)$. This situation can occur at every subtree rooted at a level-1 node. Because of the initial balancing, in one subtree we can have at most $2(d-1)$ extra steps while in all of the remaining ones we can have a total of at most $(d-1)^2$ extra steps. We conclude that the total number of extra routing steps due to class-2 packets is d^2-1. Thus, the routing of Step 6 will terminate within $n+d^2-1$ steps. ∎

Note. See [8] for a more careful refinement of Step 6 of algorithm RCdT that requires $n+2d$ steps.

Theorem 6. *Algorithm RCdT routes in an off-line fashion a permutation on an n-node complete binary tree of in at most $(1+\frac{1}{d^2-1})n+o(n)$ steps.*

In [8], we stretch the method to its limits by firstly routing packets into their destination subtrees of $O(\sqrt{n})$ nodes. Algorithm *Fast_RCdT* is developed and we prove that:

Theorem 7. *Algorithm* Fast_RCdT *routes in an off-line fashion a permutation on a complete n-node d-ary tree in at most $n+o(n)$ steps.*

6 Routing on Arbitrary Trees

The design of recursive algorithms for a complete d-ary tree was facilitated by its property that the subtrees rooted at the children of its root have the same number of nodes. This property does not hold for arbitrary trees but, fortunately, we can identify a node which possesses a similar property that makes the design of recursive algorithms possible. The following theorem is considered to be part of the folklore in graph theory.

Theorem 8. *In any n-node tree T there exists a node r such that each tree in the forest resulting by removing r (and its adjacent edges) from T has at most $\lfloor \frac{n}{2} \rfloor$ nodes.*

Given the above theorem, it is quite easy to describe the algorithm of Alon, Chung and Graham [2, 3]. In their algorithm, they firstly locate a node r satisfying the property of Theorem 8 and then they assume that the tree, say T, is rooted at r. Let r have l children, denoted by r_i, $1 \leq i \leq l$, and let T_{r_i}, $1 \leq i \leq l$, be the subtrees rooted at children of r. We introduce some new notation here. Consider a packet that is currently at a node in subtree T_{r_i}, $1 \leq i \leq l$. If the destination of the packets does not belong to T_{r_i}, we say that the packet is an *improper* packet. Otherwise, we say that it is a *proper* packet. By $I(T_{r_i})$ and $P(T_{r_i})$ we denote the set of improper and proper packets at the beginning of the routing, respectively, in subtree T_{r_i}, $1 \leq i \leq l$. The algorithm of Alon, Chung and Graham contains three phases of routing. During the first phase they route the packets such that the improper packets in each subtree form a partial tree rooted at the root of the subtree. During the second phase the route all improper packets to their destination subtrees. The routing is then completed recursively.

Given a subtree T', Alon, Chung and Graham describe a greedy method to construct a partial tree rooted at its root within $|T'|$ steps (see [2, 3] for details). By realizing that the heap-ordering algorithm described in this paper can be also used to achieve the same goal, we conclude that the routing of the first phase on subtree T' can be completed within $\min(|T'|, 2h(T'))$ routing steps. The routing of the second phase is described in the proof of Theorem 1. Given the above, it is not difficult to show that the routing can be completed within $3n$ routing steps for an n-node tree (see [2, 3] for details). In this section, we show that it is possible to (partially) overlap in time the first two phases of their algorithm. By modifying their algorithm in this way, we first improve the upper bound for the routing number of an arbitrary bounded-degree tree from $rt(T) = 3n$ to $2n + o(n)$.

In the description of the algorithm we assume that a node r satisfying the property described in Theorem 8 has been already located and that tree T has been partitioned in subtrees T_{r_i} rooted at r_i, $1 \leq i \leq l$ (Figure 3).

Algorithm *Bounded_Degree_Tree_Routing*(T) /* *BDTR* for short */

1. $m = \max_{1 \leq i \leq l} |P(T_{r_i})|$
2. **Repeat** for m routing steps:
   For each subtree T_{r_i}, $1 \leq i \leq l$, **do in parallel**
   move improper packets in T_{r_i} toward the root r_i;
3. **While** there still exist improper packets **do**
   begin
   (a) **If** the packet p currently at r has r as its destination **then**
   let T' be a subtree containing improper packets
   else
   let T' be the subtree containing the destination of the packet currently at r.
   (b) **For** each subtree T_{r_i}, $1 \leq i \leq l$, **do in parallel**
   if $T_{r_i} = T'$ **then**
   swap the packet at r with the packet at the root of T_{r_i}
   else
   move improper packets in T_{r_i} toward its root r_i. If the packet with destination r is in T_{r_i} ensure that it is the last one to be moved out of the subtree.
   end
4. **For** each subtree T_{r_i} **do in parallel**
   Bounded_Degree_Tree_Routing(T_{r_i});

Theorem 9. *Algorithm* BDTR *routes in an off-line fashion any permutation on an n-node bounded-degree tree in at most $2n + o(n)$ steps.*

Theorem 10. *Algorithm* BDTR *(with a slight modification) routes in an off-line fashion any permutation on an n-node, maximum degree 3, tree in at most $2n$ steps.*

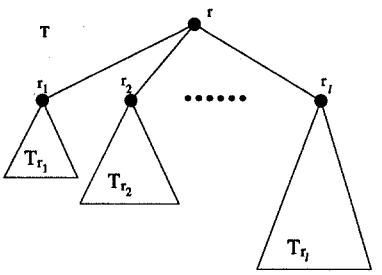

Fig. 3. Decomposing a tree T.

By incorporating the idea of partially overlapping the first two routing phases in the analysis of the algorithm of Alon, Chung and Graham [2, 3] we are also able to improve the upper bound for the routing numbers of arbitrary trees.

Theorem 11. *Any permutation on a tree with n nodes can be routed in at most $\frac{13}{5}n$ steps.*

References

1. S. Akl. *Parallel Sorting Algorithms.* Academic Press, 1985.
2. Alon, Chung, and Graham. Routing permutations on graphs via matchings. *SIAM Journal on Discrete Mathematics*, 7, 1994.
3. N. Alon, F. R. K. Chung, and R. L. Graham. Routing permutations on graphs via matchings (extended abstract). In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (San Diego, California, May 16–18, 1993)*, pages 583–591, New York, 1993. ACM SIGACT, ACM Press.
4. A. Borodin, Y. Rabani, and B. Schieber. Deterministic many-to-many hot potato routing, 1994. Unpublished manuscript.
5. N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, March 1992.
6. N. Haberman. Parallel neighbor-sort (or the glory of the induction principle). Technical Report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Road, Springfieldn VA 22151, 1972.
7. N. Rao and W. Zhang. Building heaps in parallel. *Information Processing Letters*, 37:355–358, March 1991.
8. A. Roberts, A. Symvonis, and L. Zhang. Routing on trees via matchings. Technical Report TR-494, Basser Dept of Computer Science, University of Sydney, January 1995. Available from `ftp://ftp.cs.su.oz.au/pub/tr/TR95_494.ps.Z`.
9. A. Symvonis. Optimal algorithms for packet routing on trees. In *Proceedings of the 6^{th} International Conference on Computing and Information (ICCI'94), Peterborough, Ontario, Canada*, pages 144–161, May 1994. Also TR 471, Basser Dept of Computer Science, University of Sydney, September 1993.
10. W. Zhang and R.E. Korf. Parallel heap operations on an EREW PRAM. *Journal of Parallel and Distributed Computing*, 20(2):248–255, February 1994.