Many-to-One Packet Routing on Grids

EXTENDED ABSTRACT

Yishay Mansour
Department of Computer Science
Tel-Aviv University
mansour@math.tau.ac.il

Boaz Patt-Shamir
College of Computer Science
Northeastern University
boaz@ccs.neu.edu

November 29, 1994

Abstract

We study the general many-to-one packet routing problem in a grid-topology network, where the number of packets destined at a single node may be arbitrary (in contrast to the permutation and $h-h$ routing models), and the semantics of the packets is unknown (thus disallowing packet combining). The worst case time complexity of the problem may be very high, due to the possible existence of "hot spots" in the network (regions with small boundary, which contain destinations of many packets). We therefore aim at algorithms that route all packets at the best time possible for each instance (rather than algorithms that do as well as the worst case on all instances). We present a few results for this problem. First, we give an algorithm that finds a routing schedule, such that the time to complete the routing is a constant factor away from optimal, and only $O(1)$ buffer space is needed at each node. As a by-product, the algorithm proves the interesting result that the "distance + bandwidth" lower bound is always achievable (up to constant factors). The algorithm runs in deterministic polynomial time, but it is centralized. Our second main result is a distributed on-line algorithm, which is more practically appealing. Given an instance with maximal source-destination distance D, this algorithm finds a schedule whose time complexity is at most $O(\log D)$ factor away from the best possible for that instance, using $O(\log D)$ buffer space at each node. In addition, we present an algorithm which ensures good average delivery time for the sets of packets with similar source-destination distance.

1 Introduction

It is widely believed that due to physical considerations, the ultimate model for massively parallel computers is a (two- or three-dimensional) grid of processing elements (see, for example, surveys and arguments in [BP93, FS92, Vit86]). Packet routing is one of the critical tasks that affect the performance of parallel computers dramatically. The focus of most research in packet routing on grids has been on *permutation routing*, where each node is the source and the destination of one packet. For a fixed dimensional grid, for example, it is known how to route any permutation deterministically with constant-size buffers in nearly diameter time. Another popular variant of grid routing is the $h-h$ problem, where each of the processors is the source, and the destination, of at most h packets. (An extensive survey of routing on grids can be found in [Lei91].)

While it seems reasonable to assume that each node can produce only a constant number of packets in a routing cycle, the restriction on the number of packets *destined* at a node is artificial and may require complicated coordination. The general problem, where the number of packets destined to the same node is unrestricted, is called *many-to-one routing*. Some applications deal with the naturally arising many-to-one instances by allowing the routing algorithm to "combine" packets heading for the same destination. Observe, however, that packet combining can be done only when the semantics of packet contents is known to the routing algorithm, and if this semantics allows for efficient combining. Clearly, this is only a special case. In [Lei91], Leighton comments regarding the general case that "... not much is known about optimal on-line algorithms for many-to-one packet routing on arrays [i.e., grids], and much of what is known is negative." Our paper offers a first step to remedy this state of affairs.

The pessimistic view of many-to-one routing is based on the observation that there are instances for which all routing schedules have high time complexity. More specifically, there may be a region (traditionally called a *hot spot*) with a small boundary such that many packets are destined to it, resulting in an unavoidable bottleneck. If worst-case performance is the objective to be optimized by routing algorithms, then general many-to-one-routing is hopeless. In this paper we take a more positive viewpoint of the many-to-one routing problem. Instead of dropping the problem altogether because of horrible worst cases, we suggest to strive to get the best achievable performance *for each given routing instance.* Intuitively, we would like to get flexible algorithms, which perform well on "easy" instances and not too badly on "hard" instances, or in general, algorithms that always perform close to the best possible. We remark that in the same spirit, there are several permutation routing algorithms (e.g., [RT92, Sib92]) whose time complexity depend on the largest source-destination distance in the given instance, rather than on the largest possible distance, i.e., the diameter of the network.

Our results. To evaluate the performance of a many-to-one routing algorithm on a given instance, one has not only to upper-bound the complexity of the schedules produced by the algorithm, but also to lower-bound the inherent complexity of the instance (i.e., what is the best complexity any schedule can get for the given instance). Loosely speaking, there are two basic techniques to lower bound the inherent complexity of a routing instance: length and width. The length argument, which we call the *distance* lower bound, simply says that the time to complete delivery of all packets can never be smaller than the largest source-destination distance of any packet in the given instance. The width argument, which we call the *bandwidth* lower bound, says that if n packets need to enter a region which has z incoming links, then any schedule takes at least n/z time units to deliver all packets. In many-to-one routing, hot spots induce non-trivial bandwidth lower bounds.

In this paper we show, perhaps surprisingly, that these trivial lower bounds are asymptotically tight, and present efficient algorithms which are nearly optimal. Intuitively, our results prove that "nearly full pipelining" is always possible. Specifically, given a many-to-one routing instance I, let D_I denote its distance lower bound, and let

W_I denote its bandwidth lower bound. Our first major result is the following asymptotic characterization of the many-to-one routing problem.

Theorem 1.1 *For any instance I of the many-to-one routing problem, there exists a schedule that delivers all packets in $\Theta(D_I + W_I)$ time units, using $\Theta(1)$ buffer space at each node. Moreover, such a schedule can be found in deterministic polynomial time.*

A key ingredient in the proof of Theorem 1.1 is the result of Leighton *et al.* [LMR88, LMR95], which shows that for a *given* a path for each packet, with path lengths $O(d)$ and path congestion $O(c)$ (*viz.*, at most $O(c)$ paths cross each edge), there is a schedule that delivers all packets in $O(c + d)$ time, using constant-size buffers.[1] Our main task is finding a set of paths with small d and c, which we can feed into the algorithm of [LMR95].

The algorithm used in proving Theorem 1.1 is centralized, and thus it is useful mainly in cases where the complete routing instance is known in advance. Distributed routing algorithms are more desirable in practice. Call a routing algorithm *local* if actions taken at a node depend only on the previous local state and incoming messages. Our second major result is a nearly-optimal local algorithm.

Theorem 1.2 *There exists a local algorithm for the many-to-one routing problem, such that for any instance I, all packets are delivered in $O(D_I + W_I \log D_I)$ time units, using $O(\log D_I)$ buffer space at each node.*

The algorithms described in the proofs of Theorems 1.1 and 1.2 guarantee that the *last packet* is delivered relatively quickly. Using a third algorithm, we can also guarantee good *average* delivery time on each instance, where the average is taken over sets of packets with similar source-destination distance. (Note the distinction from good *expected* behavior on a random instance.) Specifically, we have the following result.

Theorem 1.3 *Given a many-to-one instance I, let $P_I(d)$ denote the set of packets whose source-destination distance is in the interval $[2^{\lfloor \log d \rfloor}, 2^{\lfloor \log d \rfloor + 1})$. Then there exists a local algorithm such that for all $d \leq D_I$, the packets in $P_I(d)$ are delivered in average time which is at most $O(\log d \log D_I)$ factor away from the best possible, using $O(W_I \log D_I)$ buffer space at each node.*

To the best of our knowledge, the algorithms we present in this paper are the first algorithms which exhibit good behavior on any instance of the many-to-one routing problem on grids. Consider, for example, the simple technique of "greedy routing," where routes are chosen by correcting coordinates one after the other, with contentions resolved in favor of the packet with the furthest destination. It is not hard to show that there are cases in which greedy routing performs very poorly: for a k-dimensional grid, it may be the case that greedy routing completes in $\Omega(D^k)$ time, whereas an optimal solution requires only $O(D)$ time.

The remainder of this paper is organized as follows. In Section 2 we formalize the problem, state the immediate lower bound, and make a preliminary simplification regarding the size of the graph. In Section 3 we prove Theorem 1.1. In Section 4 we prove Theorem 1.2. In Appendix E we prove Theorem 1.3, using a variant of the algorithm used in the proof of Theorem 1.2. Appendices A–D contain additional definitions, proofs and constructions which were omitted from the main body of this paper due to lack of space.

2 Preliminaries

In this section, we define formally the problem we consider in this work. After defining some notation, we make a simplifying assumption regarding the size of the grid, and conclude with the basic lower bound on routing time.

[1] In [LMR88], the existential result is proven, using the Lovasz Local Lemma. In [LMR95], a polynomial-time algorithm is given, using constructive versions of the Lemma.

Problem Statement. We are given a graph $G = (V, E)$, where nodes model processors, and edges model bidirectional communication links. G is assumed to be a k-dimensional grid for some fixed constant k. See Appendix A for the standard definitions of *grid*, *subgrid* and *side-length* of a given dimension.

A *routing instance* is defined by a set of *packets* P, and two functions $source$ and $dest$, which map P to V. The *source* of a packet p is $source(p)$, and its *destination* is $dest(p)$. We assume that $source$ is injective, i.e., each node is the source of at most one packet. There is no restriction on the destination mapping $dest$.

The packet routing problem is to find a *schedule* for any given routing instance. A schedule defines an admissible way to route all packets from their source to their destination. Intuitively, the admissibility conditions below state that executions progress in synchronous steps, where packets start at their sources and arrive at their destinations by crossing links, subject to the constraint that at most one packet may cross each link in each direction at each step. Formally, a schedule is a function loc that maps packets and non-negative integers (denoting time) to nodes, such that the following conditions are satisfied.

(1) For each p, $loc(p, 0) = source(p)$, and there exists some T_p, such that $loc(p, T_p) = dest(p)$.

(2) For all p and t, if $loc(p, t) \neq loc(p, t + 1)$, then $(loc(p, t), loc(p, t + 1))$ is an edge of G.

(3) For all p and t, if $loc(p, t) \neq loc(p, t + 1)$, then there is no $p' \neq p$ such that $loc(p', t) = loc(p, t)$ and $loc(p', t + 1) = loc(p, t + 1)$.

The *delivery time* of a packet p is the minimum T such that $loc(p, T) = dest(p)$. The maximal delivery time over all packets is the *time complexity* of the schedule, denoted T_{loc}. The *load* of node v at time t is $L_v(t) = |\{p : loc(p, t) = v, dest(p) \neq v\}|$, i.e., $L_v(t)$ is the number of packets still in transit stored in v at time t. The *space complexity* of a schedule is $\max_{v,t}(L_v(t))$.

More Definitions and Notation. Throughout this paper, we use the following concepts. Let $H = (V_H, E_H)$ be any subgraph of G, and let I be a routing instance.

- H is a *line* if $V_H = \{(i_1, \ldots, i_j, \ldots, i_k) \ : \ m \leq i_j \leq M\}$, for some fixed integers $i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k$ and $m \leq M$. (This defines a line of length $M - m$ along the jth dimension.)
- H is a *cube* if it is a grid with equal side-lengths.
- $V(H) = |V_H|$, the number of nodes in H.
- $Z(H) = |\{(v, u) \in E : v \in V_H, u \notin V_H\}|$, the number of edges of G with exactly one endpoint in H.
- $N_I(H) = |\{p : dest(p) \in V_H\}|$, the number of packets of I with destination in H.
- $D_I = \max_{p \in P}(dist(source(p), dest(p)))$, the maximal source-destination distance of packets in I.
- $W_I = \max_H(N_I(H)/Z(H))$, where H ranges over all subgraphs of G.

(We shall omit subscripts when the context is clear.)

A preliminary simplification. We make the simplifying assumption that given a routing instance I, D_I is known. It is not hard to see that under this condition, we may assume w.l.o.g. that the given grid G is partitioned into cubes of side-length D, such that all packets destined at a cube are stored in that cube. This assumption amounts to a transformation, which can be done by a local algorithm at the cost of a constant factor blowup in the time complexity of the schedule. The transformation is described in Appendices B and C.

In the remainder of this paper, we therefore consider many-to-one routing instances on a single k-dimensional cube $\hat{G}$ of side-length D (whose size is $(D + 1) \times \cdots \times (D + 1)$).

We conclude this section with the basic lower bound on the time complexity of schedules.

Theorem 2.1 *All schedules* loc *for a routing instance* I *satisfy* $T_{loc} \geq \max(D_I, W_I) = \Omega(D_I + W_I)$.

3 A Proof of Theorem 1.1: An Optimal Centralized Algorithm

In this section we present an algorithm that solves the many-to-one routing problem on a cube $\hat{G}$ with side-length D. For each routing instance, the time complexity of the schedule produced by the algorithm is $O(D + W)$, and its space complexity is $O(1)$. The algorithm is centralized, and it runs in polynomial deterministic time, thus proving Theorem 1.1. Notice that Theorem 1.1, in conjunction with Theorem 2.1, shows that the algorithm is optimal, up to constant factors.

For ease of exposition, we prove the result for a two-dimensional grid; the extension to arbitrary fixed dimension is straightforward (albeit notationally cumbersome).

3.1 The Algorithm

Overview. The algorithm finds a *two-stage* schedule in three steps as follows.

1. *Find second stage paths, between the final destinations and* intermediate destinations.
2. *Find second stage schedule using the paths found at Step 1.*
3. *Find first stage schedule, from the sources to the intermediate destinations.*

In some sense, the algorithm works backwards, from the destinations to the sources. The key to the algorithm is Step 1, which finds paths with some nice properties from the final destinations to some intermediate destinations. Unlike the Valiant-Brebner scheme [VB81], the intermediate destinations are *not* random: they must satisfy certain requirements. Specifically, the set of paths chosen at Step 1 enables us, at Step 2, to complete the schedule for the second stage, so that the time complexity of the second stage is $O(D + W)$, and the intermediate destinations chosen at Step 1 are such that at Step 3, we can find an $O(D)$-time schedule for the first stage. The schedules for both stages require only $O(1)$ buffer space. The crux of our analysis is to show that such a desirable set of paths always exists. Moreover, these paths can be found efficiently, using flow techniques. Steps 2 and 3 are more standard: Step 2 is done by applying an algorithm of Leighton *et al.* [LMR88, LMR95], which finds a good timing assignment when the paths are given; and Step 3 is done using a standard optimal permutation routing as a subroutine. In the remainder of this subsection, we describe each step in more detail.

Step 1: Finding paths for the second stage. Step 1 finds intermediate destinations, and a set of paths between the final destinations and these intermediate destinations. Intuitively, our goal is that these paths will not be too long, not too congested, and that the intermediate destinations will form something close to a permutation. Formally, Step 1 produces for each packet p a path R_p such that

(1) the start point of R_p is $dest(p)$,

(2) the length of R_p is bounded by $O(D)$,

(3) the maximal number of paths over any link (called *path congestion*) is $O(W)$, and

(4) for each node v, the number of paths that end at v is bounded by $O(1)$.

This task is accomplished as follows. For $D \leq 2$, the empty paths suffice. Suppose from now on that $D \geq 3$. We create a flow network $F = (N, A)$, where the digraph F is defined as follows (see Figure 1 for a simple example).

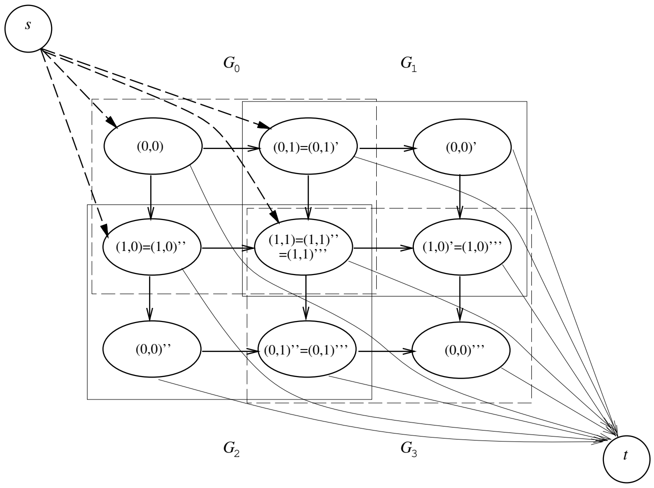

Figure 1: *The flow network F for a 2×2 cube $\hat{G}$. All extended grid nodes are connected to the sink t by edges with capacity 2 (light arrows), and the source s is connected to each node of G_0 by an edge whose capacity is the number of packets destined at that node (dashed arrows).*

We take four copies of $\hat{G}$, called $G_0, \ldots, G_3$, and "glue" them to each other as follows. Denote the nodes of G_0 by (i, j), for $0 \le i, j \le D$. Similarly, the nodes of G_1 are $(i, j)'$, the nodes of G_2 are $(i, j)''$, and the nodes of G_3 are $(i, j)'''$. Now, for all $0 \le i \le D$, we identify the nodes

$$\begin{array}{rclcrcl} (i, D) & = & (i, D)' & \qquad & (D, i)''' & = & (D, i)' \\ (D, i) & = & (D, i)'' & & (i, D)''' & = & (i, D)'' \end{array}$$

We call this part of F the *extended grid.* The edges of the extended grid are oriented as follows. All edges of G_0 are oriented from (i, j) to $(i, j + 1)$ and to $(i + 1, j)$, the edges of G_1 are oriented from $(i, j)'$ to $(i, j - 1)'$ and to $(i + 1, j)'$, the edges of G_2 are oriented from $(i, j)''$ to $(i, j + 1)''$ and to $(i - 1, j)''$, and the edges of G_3 are oriented from $(i, j)'''$ to $(i, j - 1)'''$ and to $(i - 1, j)'''$. (In Figure 1, all extended grid edges point right or down.) The way we determine the capacity of the extended grid edges is described later. Beside the extended grid, F also contains a *source* node s and a *sink* node t. The source s has only outgoing edges, one for each node of G_0. The capacity of an edge (s, v) is defined to be the number of packets destined at the node corresponding to v in the given routing instance. The sink node t has an edge of capacity two incoming from each extended-grid node in F. To complete the description of F, we now specify how the capacity of extended-grid edges is determined.

Let $F(u)$ denote the network F with capacity u assigned to each extended-grid edge. Step 1 of the algorithm proceeds by computing a maximal flow f in $F(2^i)$, for $i = 1, 2, \ldots$, until $|f| = N(\hat{G})$. Any polynomial time max-flow algorithm can be used (e.g., [GT88]). In Section 3.2, we prove that if the capacity of the extended grid edges exceeds $2W$, then $|f| = N(\hat{G})$, and as a consequence, $O(\log(2W)) = O(\log D)$ iterations of max-flow computations are sufficient. Next, the algorithm decomposes f into $N(\hat{G})$ unit-capacity *chains*, i.e., we find $N(\hat{G})$ directed paths from s to t, such that the number of paths crossing any edge e is exactly $f(e)$ (all paths are simple since F is acyclic). Chain decomposition can be done in linear time using the greedy algorithm (see, e.g., [AMO93]). Finally, the chains in F are used to obtain paths in the original graph $\hat{G}$, by mapping nodes $(i, j), (i, j)', (i, j)'', (i, j)'''$ in F to the original node $\widehat{(i, j)}$ in $\hat{G}$ and eliminating cycles. It is straightforward to verify that conditions (1–4) hold for the resulting set of paths: (1) assign to each packet p a distinct path that starts at $dest(p)$; (2) the length of any path is at most $4D$, since this is the length of the longest path in the extended grid; (3)

the path congestion is at most $O(W)$, since each edge of $\hat{G}$ hosts four extended grid edges, and each extended grid edge is part of at most $u < 4W$ chains (the extra 2 factor is due to the repeated doubling procedure); and finally, (4) the number of packets for which the same node serves as an intermediate destination is at most 8, since at most two chains end at each extended grid node. The proof that capacity of $2W$ for extended grid edges is sufficient to guarantee the existence of a distinct path for each packet (i.e., that $|f| = N(\hat{G})$) is more involved, and we dedicate Theorem 3.1 in Section 3.2 to it.

Step 2: Finding a schedule for the second stage. In the second stage of the schedule, all packets are to be routed along the paths found in Step 1. To do that, we use the powerful result of Leighton *et al.* [LMR88, LMR95]. Specifically, given a set of paths with maximal path length d and path congestion at most c, the algorithm of [LMR95] finds, in deterministic polynomial time, a schedule whose time complexity is $O(d+c)$ and whose space complexity is $O(1)$.[2] By Properties (2,3) of the paths generated at Step 1, we have that $d = O(D)$ and $c = O(W)$, and hence we get a schedule whose time complexity is $O(D+W)$. (We stress again that the algorithm of [LMR95] finds only timing assignments for the packets, when the paths for the packets are given as input.)

Step 3: Finding a schedule for the first stage. In the first stage of the schedule, packets are routed from their sources to their intermediate destinations. Recall that by Property (4) of the paths generated at Step 1, each node is an intermediate destination for only $O(1)$ packets. Hence, the goal of the first stage of the schedule is to route "nearly-a-permutation." This problem can be solved using any $O(D)$-time, $O(1)$-space permutation routing algorithm (e.g., [Kun91]), by partitioning the set of packets into $O(1)$ sets, such that in each set only a partial permutation needs to be solved (see Appendix C for a similar construction). This way, the first stage is completed in $O(D)$ steps, using $O(1)$ buffer space at each node.

Remarks.

1. The algorithm described above can be easily extended to deal with *many-to-many* packet routing instances, where each node may be the source of many packets. For example, when the total number of packets is $O(V(\hat{G}))$, a many-to-many instance can be solved by doing Steps (1–2) twice: from the sources to an intermediate "nearly-permutation," and from the destinations to another "nearly-permutation." The two intermediate configurations are then connected using a permutation routing algorithm. The resulting schedule is optimal (under an extended definition of the bandwidth lower bound, which includes the possiblity of hot-spots due to sources).

2. The reader should be aware to the fact that algorithms for the *quickest transshipment problem* can be used within our paradigm (i.e., routing the packets to an intermediate configuration of "nearly a permutation" and then apply a permutation routing algorithm). Specifically, algorithms for quickest-transshipment can be used to replace Steps 1 and 2 of our algorithm: either the classical pseudo-polynomial algorithm [FF62] (which runs in polynomial time in our case), or the recent polynomial-time algorithm of Hoppe and Tardos [HT95].

The reader should also note, however, that even though quickest transshipment algorithms yield the best possible time for the first stage of our schedule, it is unknown how to *characterize the optimal transshipment time* using only the distance and bandwidth bounds. Thus, our result is the first to establish the tight $O(D+W)$ upper bound on the time—for the special case of grids. (As an aside, note also that quickest transshipment algorithms do not guarantee constant size buffers.)

[2] In [LMR88], the existence of such a schedule is proved, and in [LMR95], an algorithm is presented. Consequently, a non-constructive version of Theorem 1.1 can be proven using only [LMR88].

3.2 The Max-Flow Value of F

The key to the correctness of the algorithm (and of Theorem 1.1) is the following theorem. Recall that $F(u)$ is the network F as described above, with capacity u assigned to each extended grid edge.

Theorem 3.1 *For $u \geq 2W$, the maximal flow value of $F(u)$ is $N(\hat{G})$.*

To prove the theorem, we first introduce a few definitions. Let A be a subset of the extended-grid nodes.

- For a node $v \in A$, $\hat{v}$ is its corresponding node in $\hat{G}$ (e.g., $(i,j)'$ is mapped to (i,j) in $\hat{G}$).
- $\hat{A} = \{\hat{v} : v \in A\}$.
- $N(A) \stackrel{\text{def}}{=} N(\hat{A})$, the total number of packets destined at the nodes corresponding to A in $\hat{G}$.
- $Z_{out}(A)$ is the number of (directed) extended-grid edges (v, v') with $v \in A$ and $v' \notin A$.

Recall that $Z(A)$ is the number of edges with one endpoint in A and the other not in A, regardless of orientation. We have the following simple lemma, relating the "undirected" $Z(\cdot)$ with the "directed" $Z_{out}(\cdot)$.

Lemma 3.2 *For any set A of nodes of G_0, $Z_{out}(A) \geq Z(A)/2$.*

Proof: We count the total contribution of each line to $Z(A)$ and to $Z_{out}(A)$. Embed the extended grid in an infinite grid whose edges are oriented consistently with the extended grid, and consider traversing a line of the infinite grid in the direction of the edges. Since the nodes of A are in G_0, A does not intersect the bottom and the right sides of the extended grid, and therefore each edge on the line that exits A contributes one unit to $Z_{out}(A)$. On the other hand, the contribution of the line to $Z(A)$ is at most the total number of edges on the line that either exit A or enter A. The proof is completed by observing that since the extended grid is finite, the number of times the infinite line enters nodes in A equals the number of times the line exits nodes in A. ∎

The following key lemma is proved after the proof of Theorem 1.1.

Lemma 3.3 *Let H be a set of nodes of the extended grid, and let H_0 be the set of nodes of G_0 in H. If $Z_{out}(H) < Z_{out}(H_0)$, then $|H| \geq D^2$.*

We are now in position to prove the theorem.

Proof of Theorem 3.1: Let $F = F(u)$ for $u \geq 2W$. By the Max-Flow Min-Cut Theorem, it is sufficient to prove that the minimum capacity of *s-t* cuts in F is $N(\hat{G})$. Note that the capacity of the cut $(\{s\}\,; F - \{s\})$ is, by construction, exactly $N(\hat{G})$. We now prove that the capacity of any *s-t* cut in F is at least $N(\hat{G})$.

Let $(S;T)$ be an arbitrary *s-t* cut of F, and denote by S_0 the intersection of S with the nodes of G_0. Consider cut edges outgoing from the source. By construction, the total capacity of these edges is $N(\hat{G}) - N(S_0)$. If S_0 is empty, we are done. So assume that S_0 is not empty; it is sufficient to show that the total capacity of the remaining edges in $(S;T)$ is at least $N(S_0)$. To prove this claim, we consider two cases.

Case 1: $Z_{out}(S - \{s\}) \geq Z_{out}(S_0)$. Clearly, using the definition of W, we have that $W \geq N(S_0)/Z(S_0)$, or equivalently $Z(S_0) \geq N(S_0)/W$. Hence, by the assumption of Case 1 and by Lemma 3.2, we have that $Z_{out}(S - \{s\}) \geq Z_{out}(S_0) \geq Z(S_0)/2 \geq N(S_0)/2W$. Thus, the total capacity contributed to $(S;T)$ by extended grid edges is at least $2W \cdot Z_{out}(S - \{s\}) \geq 2W \cdot N(S_0)/2W = N(S_0)$, and we are done for this case.

Case 2: $Z_{out}(S - \{s\}) < Z_{out}(S_0)$. By Lemma 3.3, we have that $|S - \{s\}| \geq D^2$. Each node in $S - \{s\}$ contributes two units to the capacity of the cut by virtue of its edge to the sink t, and hence we are done, because $N(S_0) \leq (D+1)^2 \leq 2D^2$ for $D \geq 3$. ∎

Proof of Lemma 3.3: We lower-bound the number of nodes in H, by bounding the number of nodes in H which are nodes of G_1, G_2 and G_3. Define the sets $L_c = \{(i,D) \in H : (i, D-1)' \in H\}$, and $L_r =$

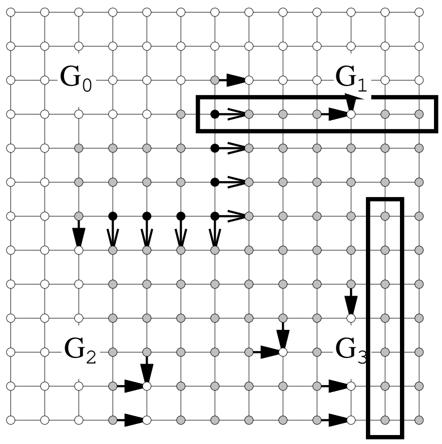

Figure 2: *Proof of Lemma 3.3. The shaded nodes are in H, and black nodes are in L_c or L_r. Full arrows represent edges contributing to $Z_{out}(H)$, and light arrows represent edges contributing to $Z_{out}(H_0)$ but not to $Z_{out}(H)$. The horizontal box shows a row of G_1, and the vertical box shows a full column of G_3.*

$\{(D, i) \in H : (D-1, i)'' \in H\}$, i.e., L_c is a subset of the rightmost column of G_0, which consists of the nodes of H in G_0 with edges to nodes of H in G_1; L_r is the analogous subset of nodes at the bottom row of G_0 (see Figure 2). Next, let l_c denote the number of rows of G_1 which intersect L_c and are completely contained in H, and let l_r denote the number of columns of G_2 which intersect L_r and which are completely contained in H. Note, for later reference, that by definitions,

- the number of nodes of H in G_1 is at least $l_c(D+1)$, and
- the number of nodes of H in G_2 is at least $l_r(D+1)$.

We now turn to bound the number of nodes of H in G_3. First, we argue that

$$Z_{out}(H - H_0) \;<\; |L_c| + |L_r| \tag{1}$$

To prove Eq. (1), note that by definition, $L_c \cup L_r$ is the set of nodes which contribute to $Z_{out}(H_0)$ and not to $Z_{out}(H)$, and hence we have $Z_{out}(H) = Z_{out}(H_0) - (|L_c| + |L_r|) + Z_{out}(H - H_0)$. Eq. (1) now follows from the assumption that $Z_{out}(H) < Z_{out}(H_0)$. Next, we argue that

$$Z_{out}(H - H_0) \;\geq (|L_c| + |L_r|) - (l_c + l_r) \tag{2}$$

Eq. (2) is proven as follows. Each row of G_1 which intersects L_c contributes at least one unit to $Z_{out}(H - H_0)$, unless it is completely contained in H; thus, the rows of G_1 account for total contribution of at least $|L_c| - l_c$ units to $Z_{out}(H - H_0)$. Similarly, at least $|L_r| - l_r$ units are contributed to $Z_{out}(H - H_0)$ by the columns of G_2, completing the proof of Eq. (2).

Together, Eqs. (1,2) imply that $l_c + l_r > 0$, i.e., there must be a complete row of G_1, or a complete column of G_2, which intersects $L_c \cup L_r$ and is contained in H. Assume w.l.o.g. that there is a row of G_1 which is contained in H, and consider now the columns of G_3. Call a column of G_3 *full* if it is completely contained in H. Observe that the existence of a complete row of G_1 in H implies that each non-full column of G_3 contributes at least one unit to $Z_{out}(H - H_0)$. Since we have already accounted for at least $(|L_c| + |L_r|) - (l_c + l_r)$ units of $Z_{out}(H - H_0)$, we get from Eq. (1) that the number of non-full columns of G_3 must be less than $l_c + l_r$. Therefore, we have that

- the number of nodes of H in G_3 is more than $(D+1)(D+1-(l_c+l_r))$.

Finally, we bound $|H|$ by adding the bounds we obtained for G_1, G_2 and G_3: using the inclusion-exclusion principle (at most $2(D+1)$ nodes may be counted twice, and at most one node may be counted three times), we conclude that $|H| > (D+1)^2 - (2D+2)$, i.e., $|H| \geq D^2$, as desired. ∎

4 A Nearly-Optimal Local Algorithm

In this section we describe and analyze a local algorithm whose time complexity is $O(D + W \log D)$, and whose space complexity is $O(\log D)$, thus proving Theorem 1.2. The basic idea of the algorithm is to apply a recursive procedure, that partitions the routing problem on a k-dimensional cube of size $D_\ell \times \cdots \times D_\ell$ into 2^k independent routing problems on subcubes of size $D_\ell/2 \times \cdots \times D_\ell/2$ each. The main difficulty in implementing this simple idea is to keep the space requirement small. To this end, one has to be careful when choosing the input to the recursive calls and their precise timing. (To gain some intuition regarding these subtleties, the reader is referred to Appendix E, where a simpler, closely related procedure with worse space complexity is presented.)

Before we describe the algorithm, let us explain the basic notion of hierarchical partition into subcubes. To avoid trivial complications, assume that the size of the given grid $\hat{G}$ is $D \times \cdots \times D$, where D is a power of 2. We hierarchically partition $\hat{G}$ (which is a k-dimensional cube) into subcubes in the following natural way: the only level 0 cube is $\hat{G}$, and each level $\ell - 1$ subcube, for $0 < \ell \leq \log D$, is partitioned into 2^k level ℓ cubes, each of size $D/2^\ell \times \cdots \times D/2^\ell$. We denote $D_\ell = D/2^\ell$.

4.1 The Algorithm

The algorithm consists of a recursive procedure called rec_route activated in cubes. The input to rec_route at cube H is a set of packets that are currently located in H, $O(1)$ packets per node, where the destinations of all packets are in H. When the procedures returns, all packets in the input set have been routed to their destinations. The important property of rec_route is that its execution time, without the recursive calls, is linear in the side length of H, regardless of the input set; efficiency is maintained by carefully managing the calls so that the number of packets in the input set of each invocation at a subcube H is $\Theta(V(H))$.

The rec_route procedure, when activated at level $\log D$ (i.e., a single node) returns immediately. When activated at level $\ell < \log D$, rec_route works in the following general structure.

1. *Route input packets to the level $\ell + 1$ subcubes of their destinations.*
2. *Do in parallel, at each level $\ell + 1$ subcube:*
   - 2.1 *Sort packets by destination address.*
   - 2.2 *Partition packets into* batches.
   - 2.3 *Repeatedly, call rec_route with a single batch as input, until all batches are done.*
3. *Return.*

Let us now elaborate some more about each step.

Step 1: Approximate routing. In Step 1, the input is a set of $O(1)$ packets per node, on a cube of size $D_\ell \times \cdots \times D_\ell$, and all the packets destinations are in the cube. When Step 1 terminates, each packet of the input is stored in one of the 2^k subcubes (of size $D_\ell/2 \times \cdots \times D_\ell/2$), which contains its destinations. In Appendix B, we describe how to do this in $O(D_\ell)$ time using $O(1)$ buffer space. In rec_route, the algorithm forces all app_route invocations at level ℓ to take the same fixed predetermined amount of time.

Step 2.1: Sorting. The sorting in Step 2.1 is done according to the following order.

Definition 4.1 *Let $v = (c_1, \ldots, c_k)$ be a node of a given k-dimensional cube of side length 2^m, and let the binary representation of each c_i be $c_i = \sum_{j=0}^{m-1} b_{i,j} 2^j$. The* linearized address *of v, denoted $R(v)$, is the km-bits binary number obtained by first listing all k most significant bits of the coordinates, then all k second most significant bits of the coordinates and so forth, ending with the k least significant bits of the coordinates. Formally:*

$$R(v) = \sum_{j=0}^{m-1} \sum_{i=1}^{k} b_{i,j} 2^{jk+(i-1)}$$

When sorting is invoked, each node stores $O(1)$ input packets. Upon termination of sorting, the number of packets stored at different nodes differ by at most 1, and each node contains a consecutive segment of the total order. In other words, if $R(v) \leq R(u)$, and by the end of the sorting p is stored at v and q is stored at u, then $R(dest(p)) \leq R(dest(q))$. Using minor modifications of standard sorting techniques, Step 2.1 can be done in $O(D_\ell)$ time units, using $O(1)$ buffer space at each node (see Appendix D.1). Similarly to Step 1, we force the sorting procedure to take a fixed predetermined amount of time.

Step 2.2: Partition into batches. Sorting induces a partition into batches as follows. At each node, the algorithm maintains "global" variables called $h(\ell)$, for $\ell = 1, \ldots, \log D$. The variables $h(\ell)$ are global in the sense that they are set once (for each ℓ), and then they retain their values throughout the execution, at all subsequent invocations. A variable $h(\ell)$ is set when the first sorting terminates in the level ℓ subcube in which the node resides; the value it takes is the maximal load after sorting, over all nodes in that subcube (recall that after sorting, the load at different nodes differ by at most 1). Given $h(\ell+1)$, the partition into batches at Step 2.2 is done by picking every "$h(\ell+1)$-th packet" in the sorted order: for a packet p, let $s(p)$ be the number of packets with linearized destination address smaller than $R(dest(p))$. Then for $j = 1, \ldots h(\ell+1)$, batch j consists of all packets p such that $s(p) \equiv j - 1 \pmod{h(\ell+1)}$.

Step 2.3: Recursive calls. The rec_route procedure is invoked recursively at a level $(\ell+1)$-st subcube for $h(\ell+1)$ consecutive times (all nodes have the same $h(\ell+1)$ value), where the input for the j-th invocation is the j-th batch. We prove later that each batch contains $O(1)$ packets per node.

Step 3: Termination. A level ℓ invocation of rec_route returns only after all its recursive calls have returned, at all its level $(\ell+1)$-st subcubes.

4.2 Analysis

In this section we sketch the analysis of the algorithm specified in Section 4.1. We only state the main lemmas as milestones; more detailed analysis can be found in Appendix D.2.

We start with a few definitions. Define *execution tree* be the tree whose nodes are invocation of rec_route, such that a node a is a parent of a node b iff the invocation corresponding to a called the invocation corresponding to b. In the analysis that follows, we say that a "subcube H was invoked" and "rec_route was invoked in H" interchangeably. Relying heavily the properties of our sorting and the definition of the $h(\ell)$ variables, we can prove an upper bound on the number of packets in an invocation in the following key lemma.

Lemma 4.1 *The number of packets in each invocation of the algorithm at subcube H is at most $2V(H)$.*

Call an invocation *active* at a given time if at that time, it has been called, hasn't returned yet, and the control is not at a recursive call it spawned. From Lemma 4.1 we get the following corollary.

Corollary 4.2 *Any level ℓ invocation is active $O(D_\ell)$ time units.*

Next, we define the *timing behavior* of an invocation. Intuitively, timing behavior describes the way the execution subtree rooted at the given invocation evolves in time. Formally, it is a mapping that assigns to every node of the tree the time in which the corresponding invocation started, where the root invocation is mapped to 0. Since we force steps 1 and 2.1 of each invocation to take the same amount of time in each invocation, and since the h variables have fixed values throughout the execution, we get that all invocations of the same subcube have the same timing behavior. This nice regularity of the structure of executions gives rise to the concept of *critical path*, defined as follows.

Definition 4.2 *The* critical path *of a given execution of the algorithm is a sequence of subcubes* $P_0, P_1, P_2, \ldots, P_{\log D}$*, where* P_0 *is the level* 0 *cube, and for* $\ell > 0$*,* P_ℓ *is the subcube of level* ℓ *that returns last at the invocation of* $P_{\ell-1}$ *(ties are broken arbitrarily).*

The following lemma says that it is sufficient to analyze the time in which critical path subcubes are active.

Lemma 4.3 *At any point in the execution, one of the subcubes of the critical path is active.*

Before we proceed, we introduce a few additional definitions.

- $P_1, \ldots, P_{\log D}$ is the sequence of critical path subcubes, where P_ℓ is in level ℓ.
- m_ℓ is the total number of times P_ℓ was invoked, i.e., $m_0 = 1$, and $m_{\ell+1} = m_\ell h(\ell + 1)$.
- B_ℓ is the maximum of the average number of packets destined at $P_0, \ldots, P_\ell$. Formally, $B_0 = 1$, and $B_{\ell+1} = \max(B_\ell, N(P_{\ell+1})/V(P_{\ell+1}))$.

Using the above concepts, we have the following upper bound for the total number of invocations of a subcube on the critical path.

Lemma 4.4 *For all* ℓ*,* $m_\ell \leq 2B_\ell$*.*

Finally, we state the following simple fact.

Lemma 4.5 *For all* ℓ*,* $B_\ell D_\ell = O(W + D_\ell)$*.*

We can now prove Theorem 1.2. We first restate it.

Theorem 1.2 *There exists a local algorithm for the many-to-one routing problem, such that for any instance* I*, all packets are delivered in* $O(D_I + W_I \log D_I)$ *time units, using* $O(\log D_I)$ *buffer space at each node.*

Proof: The space upper bound follows from the observation that at each node, in each time step, there are at most $1 + \log D$ levels that haven't returned, and each level incurs $O(1)$ load.

To get a time upper bound, note that by Lemma 4.3, it is sufficient to bound the number of time steps in which subcubes of the critical path are active. Let T denote the total time in which critical subcubes are active. By Corollary 4.2, each P_ℓ is active $O(D_\ell)$ time units at any of its invocations. Therefore, by the definition of m_ℓ, Lemma 4.4, and Lemma 4.5, we get that

$$T \leq \sum_{\ell=0}^{\log D} m_\ell O(D_\ell) \leq \sum_{\ell=0}^{\log D} O(B_\ell D_\ell) \leq \sum_{\ell=0}^{\log D} O(W + D_\ell) = O(W \log D + D)\,. \quad \blacksquare$$

Acknowledgments

We thank Franco Preparata and Gianfranco Bilardi for helpful discussions, and we thank Tom Leighton and Éva Tardos for providing us with preprints of their manuscripts. In particular, we thank Tom Leighton for detecting an error in one of our early algorithms.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice-Hall, Engelwood Cliffs, New Jersey, 1993.

[BP93] Gianfranco Bilardi and Franco Prepaprata. Horizons of parallel computation. Research Report CS-93-20, Department of Computer Science, Brown University, May 1993.

[FF62] Lester R. Ford, Jr. and D. Ray Fulkerson. *Flows in Networks*. Princeton U. Press, 1962.

[FS92] Yosee Feldman and Ehud Shapiro. Spatial machines: A more realistic approach to parallel computation. *Comm. ACM*, 35(10):61–73, 1992.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *J. ACM*, 35(4):921–940, October 1988.

[HT95] Bruce Hoppe and Éva Tardos. The quickest transshipment problem. In *Proc. of the 6th ann. ACM-SIAM Symposium on Discrete Algorithms*, January 1995. To appear.

[Kun91] Manfred Kunde. Concentrated regular data streams on grids: sorting and routing near to the bisection bound. In *32nd Annual Symposium on Foundations of Computer Science*, pages 141–150, October 1991.

[Lei91] Tom Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan-Kaufman, 1991.

[LMR88] Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 256–269, White Plains, NY, October 1988.

[LMR95] Tom Leighton, Bruce Maggs, and Satish Rao. Fast algorithms for finding O(congestion+dilation) packet routing schedules. In *Proc. 28th Hawaii International Conference on System Sciences*, Januray 1995. To appear.

[RT92] Sanguthevar Rajasekaran and Thanasis Tsantilas. Optimal routing algorithms for mesh-connected processor arrays. *Algorithmica*, 1992(8):21–38, 1992.

[Sib92] Jop Sibyen. *Algorithms for routing on meshes*. PhD thesis, Department of Computer Science, Utrecht University, 1992.

[VB81] Leslie G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, Milwaukee, WI, May 1981.

[Vit86] Paul M.B. Vitanyi. Nonsequential computations and the laws of nature. In *Aegian Workshop on Computing*, pages 108–120, Berlin, 1986. Springer-Verlag (LNCS 227).

APPENDICES

A Grid Definitions

In this section we provide some definitions for the concept of grid graphs.

A graph is a k *dimensional grid* if it is isomorphic to $G = (V, E)$, where there exist non-negative integers $n_1, \ldots, n_k$, such that

$$V = \{0, \ldots, n_1\} \times \cdots \times \{0, \ldots, n_k\} \ ,$$

and $((i_1, \ldots, i_k), (j_1, \ldots, j_k)) \in E$ iff $\sum_{l=0}^{k} |i_l - j_l| = 1$.

The *size* of the grid is $(n_1 + 1) \times \cdots \times (n_k + 1)$. The *side length* of G in the lth dimension is the number n_l. A *cube* is a grid with all side lengths equal. A *subgrid* is a subgraph which is a grid.

B Approximate Routing

In this section we define a parallel procedure called app_route. The app_route task is invoked at a cube of size $D_\ell \times \cdots \times D_\ell$ (i.e., side length $D_\ell - 1$), which is naturally partitioned into 2^k subcubes of side length $D_\ell/2$ (of side length $D_\ell/2 - 1$). The input to app_route is a set of packets with maximal initial load M, and destination in the given cube. When app_route terminates, each packets is stored in the subcube that contains its destination. The procedure works as follows.

The path a packet traverses consists of up to k segments, where each segment "corrects" one of the coordinates of the packet. Specifically, the length of each segment is exactly $D_\ell/2$. The path is defined iteratively as follows. Suppose that p is currently stored at node $(c_1, \ldots, c_k)$, and let i be the first coordinate such that c_i is not in the range of the target subcube. Then the next segment of the route is the shortest path connecting $(c_1, \ldots, c_i, \ldots, c_k)$, and $(c_1, \ldots, (c_i + D_\ell/2) \bmod D_\ell, \ldots, c_k)$.

The packets progress in k stages, where in the i-th stage, only the i-th coordinates are corrected. Conflicts over links are resolved arbitrarily.

We summarize the complexity of app_route in the following theorem.

Theorem B.1 *Given a k-dimensional grid with side length $2D_\ell$ and initial maximum load M, approximate routing can be solved in $O(M2^kD_\ell)$ time units and space complexity $O(Mk2^k)$.*

Proof Sketch: It is easy to verify, by induction on the stages, that stage i starts with at most $M2^{i-1}$ packets per node. Therefore, since in stage i there are at most $D_\ell M 2^{i-2}$ packets at any segment of length $D_\ell/2$, stage i ends in time proportional to $D_\ell M 2^i$. It follows that app_route terminates in total time $O(M2^kD_\ell)$. ∎

C Reducing a General Instance to a Cube Instance

Suppose we are given an algorithm for packet routing over a k-dimensional cube of side-length D. In this section we describe how to use such an algorithm for packet routing over arbitrary k-dimensional grids, when the maximal source-destination distance in the routing instance is D, at the expense of a constant blowup in time and space complexity.

In the given k-dimensional grid (whose size is arbitrary), we use two partitions, where each partition is composed of subcubes of size $2D\times\cdots\times2D$ The partitions have overlap of size D in each dimension. Formally, define the l *cube* *with corner* $(o_1, \ldots, o_k)$ to be the the subgraph induced by the node set $\{(o_1 + i_1, \ldots, o_k + i_k) : \forall j \;\; 0 \le i_j < l\}$. *Partition I* is defined by all the $2D$-cubes with corner $(o_1, \ldots, o_k)$, where o_j is divisible by $2D$ for all $1 \le j \le k$. Similarly, *Partition II* is defined by all the $2D$-cubes with corner $(o_1, \ldots, o_k)$, where $o_j \equiv D \bmod 2D$ for all $1 \le j \le k$. Finally, let the *canonical cubes* be the D-cubes with corner $(o_1, \ldots, o_k)$, where o_j is divisible by D for all $1 \le j \le k$.

Given a routing instance, the algorithm reduces it to independent routing instances on the canonical cubes of the main partition as follows. First, the app_route procedure (see Appendix B) is applied at the subcubes of Partition I, where its input consists of all packets whose destination is not in their current canonical cube. Then app_route is applied at the subcubes of Partition II, and its input is the remaining set of packets whose destination is not in their current canonical cube. Notice that for each packet, there exists either a subcube in Partition I or in Partition II such that the destination and the source of the packet are in the subcube. It follows that after the two applications of app_route, all packets are in the same canonical as their destination. Moreover, the number of packets stored at each node is $O(1)$. The nest step in the transformation is to apply the given algorithm (for cubes) over the canonical cubes $O(1)$ times at each cube, where each application takes at most one packet per node as input. This completes the description of the transformed algorithm.

To analyze the complexity of the transformed algorithm, notice that the preliminary stage of app_route takes $O(D)$ time and $O(1)$ space by Theorem B.1. Since the time complexity of any schedule for the given instance is $\Omega(D)$ by Theorem 2.1, we conclude that the whole transformation incurs only a constant blowup in the time and space complexity of the given algorithm for cubes of side length D.

D Proofs for Section 4

In this section we provide more details about the construction and analysis presented in Section 4.

D.1 Sorting on Grids

We use a variant of the following result.

Theorem D.1 [Lei91] *Suppose that at a given k-dimensional grid with side length D_ℓ there is one item per node, with total order defined on the items. Then the items can be sorted in $O(D_\ell)$ time units with $O(1)$ storage space per node.*

In the algorithm of Section 4.1, when sorting is invoked, each node stores $O(1)$ input packets. Upon termination of sorting, the number of packets stored at different nodes differ by at most 1, and each node contains a consecutive segment of the total order. It is straightforward to see that these modifications do not change the problem essentially. By letting each processor simulate $O(M)$ consecutive processors, we get the following corollary.

Corollary D.2 *If the number of items per node is at most M, then sorting can be done in $O(MD_\ell)$ time units, using $O(M)$ storage space in each node.*

D.2 Detailed Analysis of the rec_route Procedure

Our first step is to prove the following property of two "cousin" invocations of the same subgrid.

Lemma D.3 *Let H be a level $(\ell - 1)$-subcube that contains H' of level ℓ. Then the number of packets destined to any given subcube $H'' \subseteq H'$ in any two sibling invocations of H' differ by at most one.*

Proof: Let x and x' be any two siblings in H', and let x_0 be their common parent in H. Consider the configuration of the packets in x_0 after the sorting of H' is completed. By the definition of the linearized address, all packets destined to H'' are consecutive in the sorted order (they have identical significant bits). The result follows, since by the way batches are defined, for all children x of x_0 we have $n_{H''}(x) \in \{\lceil n(H'')/h(\ell)\rceil, \lfloor n(H'')/h(\ell)\rfloor\}$. ▮

Using Lemma D.3 we can prove the following key property of cube invocations.

Lemma D.4 *Let H be any subcube. Then the number of packets destined to any subcube $H' \subseteq H$ in different invocations of H differ by at most one.*

Proof: By contradiction. Suppose that the lemma is false, and let H_0 be a *largest* subcube for which there exist $H' \subseteq H_0$, and invocations x_1, x_2 of H_0, such that $n_{H'}(x_1) - n_{H'}(x_2) \geq 2$. Clearly, the level of H_0 is positive, since there is only one invocation of the level 0 cube. Let H_{-1} be the level $\ell - 1$ subcube that contains H_0, and let y_1 and y_2 denote the invocations that spawned x_1 and x_2 in H_{-1}, respectively. We now show that $n_{H'}(y_1) - n_{H'}(y_2) > 1$, contradicting the maximality of H_0.

This is proven as follows. By the algorithm, both y_1 and y_2 invoke H_0 exactly $h(\ell)$ times. Furthermore, from Lemma D.3 we get that all other invocations of H_0 by y_1 have at least $n_{H'}(x_1) - 1$ packets. We therefore have that $n_{H'}(y_1) \geq n_{H'}(x_1) + (h(\ell) - 1)(n_{H'}(x_1) - 1)$. A similar argument shows that $n_{H'}(y_2) \leq n_{H'}(x_2) + (h(\ell) - 1)(n_{H'}(x_2) + 1)$. These inequalities imply the desired contradiction, since we get

$$\begin{aligned} n_{H'}(y_1) - n_{H'}(y_2) &\geq n_{H'}(x_1) - n_{H'}(x_2) + (h(\ell) - 1)((n_{H'}(x_1) - 1) - (n_{H'}(x_2) + 1)) \\ &= (n_{H'}(x_1) - n_{H'}(x_2) - 2)h(\ell) + 2 \quad \geq \quad 2\,, \end{aligned}$$

since we assumed that $n_{H'}(x_1) - n_{H'}(x_2) \geq 2$. ▮

We can now prove Lemma 4.1. We first re-state it.

Lemma 4.1 *The number of packets in each invocation of the algorithm at subcube H is at most $2V(H)$.*

Proof: By induction on the levels. For the level 0 subcube, the lemma follows from the assumption that the input to the routing problem consists of no more than one packet per node. Consider now a subcube H of level $\ell > 0$, with parent subcube $H' \supset H$. Let n_1 be the maximum number of packets destined for H in invocations of H'. If $n_1 \leq 1$ we are done trivially. Suppose now that $n_1 \geq 2$. From Lemma D.4 it follows that the number of packets in the first invocation of the algorithm in H' is at least $n_1 - 1$. Hence, by the algorithm, $h(\ell) \geq \lceil (n_1 - 1)/V(H)\rceil$. By definition of n_1, we have that the number of packets in any invocation of H is at most

$$\frac{n_1}{\lceil (n_1 - 1)/V(H)\rceil} \leq \frac{n_1 V(H)}{n_1 - 1} \leq 2V(H)\,. \quad ▮$$

Lemma 4.3 *At any point in the execution, one of the subcubes of the critical path is active.*

Proof: Let the critical path be $P_0, P_1, \ldots, P_{\log D}$. We prove, by backwards induction on the levels, that at any step during the time interval that starts with the invocation of P_ℓ and ends when P_ℓ returns, P_i is active for some $\ell \leq i \leq \log D$. For $\ell = \log D$ there is nothing to prove. Consider an invocation of level ℓ. After app_route and the sorting in P_ℓ (in which P_ℓ is active), the first batch in $P_{\ell+1}$ is invoked. By induction hypothesis, at all time steps until $P_{\ell+1}$ returns, there exists $\ell + 1 \leq i \leq \log D$ such that P_i is active. Whenever $P_{\ell+1}$ returns, either P_ℓ invokes it again, or else it returns also, since by the definition of critical path, $P_{\ell+1}$ is the last invocation to return in P_ℓ. This completes the inductive step. ▮

Recall the definitions.

- $P_1, \ldots, P_{\log D}$ is the sequence of critical path subcubes, where P_ℓ is in level ℓ.

- m_ℓ is the total number of times P_ℓ was invoked, i.e., $m_0 = 1$, and $m_{\ell+1} = m_\ell h(\ell + 1)$.
- B_ℓ is the maximum of the average number of packets destined at $P_0, \ldots, P_\ell$. Formally, $B_0 = 1$, and $B_{\ell+1} = \max(B_\ell, N(P_{\ell+1})/V(P_{\ell+1}))$.

We have the following lemma.

Lemma 4.4 *For all ℓ, $m_\ell \leq 2B_\ell$.*

Proof: By induction on the levels. For $\ell = 0$ the claim trivially holds, since $m_0 = 1$ and $B_0 = 1$. For the inductive step, let $\ell > 0$, and consider the situation in $P_{\ell-1}$ (that invokes rec_route in P_ℓ). If $h(\ell) = 1$, then $m_\ell = m_{\ell-1}$, and we are done by induction, since $B_\ell \geq B_{\ell-1}$. Otherwise $h(\ell) \geq 2$. Then it follows from the definition of $h(\ell)$ that the total number of packets destined for P_ℓ in the first invocation of $P_{\ell-1}$ is at least $(h(\ell)-1)V(P_\ell)+1$. Therefore, by lemma D.4, the number of packets destined for P_ℓ in any invocation of $P_{\ell-1}$ is at least $(h(\ell) - 1)V(P_\ell)$, and hence the number of input packets at each invocation of P_ℓ is at least

$$\frac{(h(\ell) - 1)V(P_\ell)}{h(\ell)} \geq \frac{V(P_\ell)}{2} \,.$$

We may therefore conclude that

$$m_\ell \leq \frac{N(P_\ell)}{V(P_\ell)/2} \leq 2B_\ell \,,$$

and the induction is complete. ▮

Lemma 4.5 *For all ℓ, $B_\ell D_\ell = O(W + D_\ell)$.*

Proof: Notice that $B_\ell D_\ell$ can be maximized only if $\ell = 1$ (in which case $B_\ell = 1$), or else it must be the case that $B_\ell = N(P_\ell)/V(P_\ell) \leq N(P_\ell)/(Z(P_\ell)D_\ell) \leq W/D_\ell$. since $D_{\ell+1} < D_\ell$ for all ℓ. The result follows. ▮

E Good Average Delivery Time on All Instances

In this section we prove Theorem 1.3, which deals with another variant of the problem, where the objective is to guarantee average case delivery time which is close to the best possible for the given instance.

Notice that there is no way to guarantee optimality with respect to individual packets, since for any given packet there exists a schedule that delivers it in time equal to its source-destination distance (possibly at the expense of delaying other packets). Therefore, we consider the average transit time for certain sets of packets. Recall the statement of the result.

Theorem 1.3 *Given a many-to-one instance I, let $P_I(d)$ denote the set of packets whose source-destination distance is in the interval $[2^{\lfloor \log d \rfloor}, 2^{\lfloor \log d \rfloor + 1})$. Then there exists a local algorithm such that for all $d \leq D_I$, the packets in $P_I(d)$ are delivered in average time which is at most $O(\log d \log D_I)$ factor away from the best possible, using $O(W_I \log D_I)$ buffer space at each node.*

The basic idea is to handle each class independently, by using *time multiplexing*. More precisely, we present a routing algorithm that ensures good average delivery time if all packets have similar source-destination distances, and run $\log D$ independent "versions" of it, where each version deals exclusively with one class of packets. This is done by letting each of the $\log D$ versions use the communication links one time unit every $\log D$ time units, and multiplying the space requirement by $\log D$.

Our basic algorithm is the following recursive procedure, called bfs_route.

1. *Route input packets to the level $\ell+1$ subcubes of their destinations.*
2. *Do in parallel, at each level $\ell+1$ subcube:*
   2.1 Balance the load over the nodes.
   2.2 Call bfs_route with all packets as input.

The main difference between bfs_route and rec_route (specified in 4.1) is that all packets present at a subcube are sent as input in the recursive call (whereas in rec_route, the input of each invocation—called a "batch" in Section 4.1— consists of roughly one packet per node). This change eliminates the need for precise sorting, and enables us to use an elementary load-balancing procedure at Step 2.1, whose only objective is to re-distribute the packets more-or-less evenly in the subcube (their order does not matter). We sketch below the analysis of bfs_route.

First we state a simple fact for Step 2.1. Given an assignment of packets to a set of nodes S, say that S is *β-balanced* if the maximal difference in load in S is no more than β; the *imbalance* of S is β if S is $(\beta-1)$-balanced and not β-balanced. We have the following easy theorem.

Theorem E.2 *Given a cube of dimension k with side length D_ℓ and initial load imbalance at most β, k-load balancing can be performed in $O(\beta D_\ell)$ time units, with maximal load never above the initial maximal load.*

We use the following definition.

Definition E.1 *Given a subgrid H, define $A(H) = \max\{1, N(H)/V(H)\}$.*

The correctness of bfs_route is implied by the following straightforward invariant.

Lemma E.3 *Let H be a subcube of level $0 \leq \ell < \log D$. By the completion of Step 2.1 in H, all packets with destination in H are in the level $(\ell+1)$-st subcubes of their destinations, and each of the level $(\ell+1)$-st subcubes H' has maximal load $O(A(H'))$.*

Proof Sketch: By induction on levels. The base case $\ell = 0$ is obvious. For the inductive step, note that by the inductive hypothesis and Theorem B.1, we have that when Step 1 ends at a level ℓ subcube H, all packets are in their level $\ell+1$ subcubes, and the maximal load in each subcube is bounded by $O(k2^k A(H))$. After Step 2.1 is done in a level $\ell+1$ subcube H', the load in nodes of H' is reduced to $O(A(H'))$. ▮

Lemma E.3 also provides an upper bound on the space complexity of the resulting schedule: note that the maximal load may grow only by a constant factor in each iteration. This leads to the following result.

Lemma E.4 *The space complexity of the schedule is $O(W)$.*

Proof: By Lemma E.3, the space complexity is bounded by $O(\max_H A(H))$ over all subcubes H. The result follows, since for any subset H we have $A(H) = O(N(H)/V(H)) \leq O(N(H)/Z(H)) = O(W)$. ▮

The idea behind the analysis of the running time of the algorithm is the following. Call Steps 1 and 2.1 at a level ℓ subcube a *phase* ℓ of the algorithm. Note that each packet undergoes phases $0, \ldots, \log D$ one immediately after the other. (This in contrast to rec_route, where packets are waiting for other subcubes to complete). We show that each phase of the algorithm is done optimally in some sense, and thus the number of steps needed to deliver a packet is close to optimal.

Formally, we have the following lemma for the algorithm.

Lemma E.5 *Let H be a subgrid of level $\ell \geq 0$. Then phase ℓ is completed in H in $O(A(H)D_\ell)$ time units.*

Proof Sketch: By lemma E.3, the maximal load at G at the beginning of phase ℓ is $M = O(A(H)$. Hence, by Theorem B.1, Step 1 is done in $O(A(H)D_\ell)$ time units. Also, the maximal load in each of the subgrids at that point is $O(A(H))$, and hence the imbalance when the load balancing is invoked is also at most $O(A(H))$. From Theorem E.2 it follows that the load balancing stage terminates in $O(A(H)D_\ell)$ time units. ▮

We can now state the main result we need for average delivery time.

Lemma E.6 *Suppose that in a given routing instance, there exists some $d > 0$ such that for all packets p, $d \le dist(source(p), dest(p)) < 2d$. Then the average delivery time of a packet under the algorithm bfs_route above is it most $O(\log d)$ factor times the best possible average time for the given instance.*

Proof Sketch: We focus on the sum, over all packets, of "packet steps," i.e., the time each packet spent before reaching its destination. Let p be an arbitrary packet, and denote by $H_0(p), \ldots, H_{\log d}(p)$ the sequence of cubes that contain its destination, where H_ℓ is at level ℓ. By Lemma E.5, p reaches its destination in time $\sum_{\ell=0}^{\log D} O(A(H)D_\ell) = O(N(H_\ell(p))/Z(H_\ell(p)))$, and therefore the total number of packet steps "spent" by the algorithm, over all packets, is at most

$$O\left(\sum_p \sum_{\ell=0}^{\log D} N(H_\ell(p))/Z(H_\ell(p))\right) = O\left(\sum_H N(H)^2/Z(H)\right) ,$$

where H ranges over all subcubes in the hierarchical partition of $\hat{G}$.

Consider now an arbitrary schedule. First note that since the distance between the sources and destinations is at least d, it follows that for every subcube H with side length smaller than d, the number of packets crossing its boundary is at least $N(H)$. It is easy to see that the average time for a packet to cross the boundary is $\Omega(N(H)/Z(H))$; summing over all the $N(H)$ packets, we get a contribution of $\Omega(N(H)^2/Z(H))$ to the total number of packet steps.

Consider now the sum of $N(H)^2/Z(H)$ over all subcubes H with side length smaller than d. It may be the case that a single packet step is counted more than once (intuitively, a packet may "wait" simultaneously for more than one subcube boundary). However, observe that in this sum, each packet step is counted at most $\log D$ times—since a packet step is associated with at most one subcube for each level (specifically, the one that contains its destination). We conclude that the total number of packet steps spent by any schedule is at least $\Omega\left(\frac{1}{\log D}\sum_H N(H)^2/Z(H)\right)$, where H ranges over all subgrids H with side length smaller than d.

To complete the proof, note that the routing of the packets into the subgrids with side length smaller than d is completed in $O(d)$ time units, incurring at most a constant blowup to the average. ∎