An Analysis of Network Routing and Communication Latency

by

Yihao Lisa Zhang

B.A. Wellesley College, Wellesley, MA (1993)

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

Author .....
Department of Mathematics
May 9, 1997

Certified by .....
Frank Thomson Leighton
Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Hung Cheng
Chairman, Applied Mathematics Committee

Accepted by .....
Richard Melrose
Chairman, Departmental Committee on Graduate Students

An Analysis of Network Routing and Communication Latency

by

Yihao Lisa Zhang

Submitted to the Department of Mathematics
on May 9, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The thesis is divided into two parts. In the first part we describe methods for mitigating the degradation in performance caused by high latencies in parallel and distributed networks. For example, given any "dataflow" type of algorithm that runs in T steps on an n-node ring with unit link delays, we show how to run the algorithm in $O(T)$ steps on any n-node bounded-degree connected network with *average* link delay $O(1)$. This is a significant improvement over prior approaches to latency hiding, which require slowdowns proportional to the *maximum* link delay. In the case where the network has average link delay d_{ave}, our simulation runs in $O(\sqrt{d_{\text{ave}}}T)$ steps using $n/\sqrt{d_{\text{ave}}}$ processors, thereby preserving efficiency. We also show how to efficiently simulate an $n \times n$ array with unit link delays using slowdown $\tilde{O}(d_{\text{ave}}^{2/3})$ on a 2-dimensional array with average link delay d_{ave}. Lastly, we present results for the case in which large local databases are involved in the computation.

In the second part of the thesis we design schedules that provide per-packet delay guarantees in connection-oriented networks. We consider a network with arbitrary topology on which a set of sessions is defined. For each session i, packets are injected at a rate r_i to follow a predetermined path of length d_i. Due to limited bandwidth, one packet at a time may advance on a link. Packets therefore may experience end-to-end delays while traversing from their sources to their destinations. For the first time, we present an asymptotically-optimal schedule that achieves a delay bound of $O(1/r_i + d_i)$ for every session-i packet, as long as the total rates add up to less than 1 on each link. An additional bonus is that only constant queues are needed at the switches. We also describe a simple distributed algorithm that, with high probability, delivers every session-i packet to its destination within $O(1/r_i + d_i \log(m/r_{\min}))$ steps of its injection where $r_{\min}$ is the minimum session rate, and m is the number of links in the network.

Thesis Supervisor: Frank Thomson Leighton
Title: Professor of Applied Mathematics

Acknowledgements

I am grateful to many people who have made this thesis possible. First, I would like to thank my advisor Professor Tom Leighton for the all the time and effort that he has devoted to my graduate study. Tom has been a source of intriguing problems and brilliant ideas, and he has never stopped being positive and encouraging, even during setbacks in my research. The discussions with him have been invaluable to me. I feel truly fortunate to have been his student. I would also like to thank Professors Michel Goemans and Dan Kleitman to serve on my thesis committee.

The work in this thesis is joint with several other people, including Matthew Andrews, Antonio Fernandez, Mor Harchol-Balter, Tom Leighton and Takis Metaxas. Thanks to them for their kind permission to put our joint results into my thesis. I also had the fortune to collaborate with Bill Aiello, Yonatan Aumann, Michael Bender, Sandeep Bhatt, Michel Goemans and K. R. Krishnan, although the work is not included here.

I am indebted to a graduate fellowship from the National Science Foundation that funded me during the past three years and an Applied Mathematics Fellowship from MIT that funded my first year. Additional support was provided by ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246. I would also like to thank Takis for kindly sharing his Brachman-Hoffman research fellowship with me.

True friends are those who share good and bad times with me. I would like to thank Scott and Linda Dynes, Yue Hu and Juan Sanchez, Bin Hu, and Michael Bender for their friendship over the years. The Holdsworths, my adopted American family, deserve a special word of thanks. They have provided a warm, comfortable home and many memorable holidays in this foreign land. I shall always cherish their love and hospitality.

Many special thanks go to Matthew Andrews, my best friend and frequent coauthor. Together we have spent numerous weekends and late nights cranking out papers and catching deadlines. It is wonderful to collaborate with someone who is modest about himself, charitable to others and full of creative ideas. I have benefited greatly

from his intelligence, good humor, and unfailing support.

Finally, I owe my deepest gratitude to my close and loving family. My Jie-Jie is the best sister and a steady source of comfort. Without her love and understanding I would not have enjoyed my time in America as much. My father and late grandmother have always been dedicated to my well-being. Their sacrifice ensured the best education for me in China, and their vision brought me to Wellesley and then to MIT. Their love will continue to give me strength through every stage of my life. I dedicate this thesis to them.

To

My Father and Grandmother

Contents

Chapter 1

Introduction

Recent technology advances have motivated a great deal of research in the fields of computation and communication. One general problem is network *latency*, i.e. the delay experienced by information as it travels from one location to another. Since latency has a direct impact on performance, we investigate this issue in two contexts. First, we study how to hide the effect of communication delay in high performance computing. Second, we present methods that ensure fast information delivery in communication networks.

Much effort is directed towards satisfying the increasing demands of parallel and distributed computing. One line of research is devoted to massively parallel processors (MPPs) that are dedicated to parallel computing, e.g. CRAY supercomputers and Connection Machines. Networks of workstations (NOWs) provide another platform for high performance computing [3]. The idea here is to connect desktop computers that are designed for small interactive jobs over a network. In this way, NOWs are capable of acting as a distributed supercomputer. Among various other issues such as job scheduling and load balancing, communication latency determines the efficiency of computing, especially when a NOW is involved. In Part I of this thesis, we devise methods for mitigating the impact of communication latency in parallel and distributed computing.

Many experts believe that the communication networks of the future will be based on Asynchronous Transfer Mode (ATM) technology, which will enable the integra-

tion of traffic with a wide range of characteristics within a single communication network [28]. A central problem in the design of ATM networks is that of providing quality-of-service (QoS) guarantees, such as the guarantees on bandwidth, delay, jitter and packet loss. These issues are particularly important if the networks support real-time traffic such as voice and video. Good scheduling algorithms can ensure fairness among the traffic streams and the efficient use of the network resources. In Part II of this thesis, we study the scheduling disciplines for connection-oriented packet-switched networks, e.g. ATM. We design schedules that efficiently deliver packets to their destinations, thereby providing end-to-end delay guarantees.

1.1 Hiding Latency for Parallel and Distributed Computation

In Part I of this thesis, we devise methods for mitigating the impact of latency in parallel and distributed computing. We focus on a model of processors interconnected by a bounded-degree fixed-connection network. The network may be either well-structured (such as an array) or unstructured (such as an arbitrary binary tree), and it may be either real or virtual. We assume that each link (or edge) in the network has a delay which models the latency associated with using the link. We also assume that the links have sufficient bandwidth and can be pipelined.

Most papers describing algorithms for parallel or distributed computation assume a model of computation in which all the links have unit delay. Such a model is nice to work with and it is realistic for some parallel machines, but not for most. In reality, there are often substantial delays associated with some or all of the links. These delays can be caused by long wires, links that are realized by paths that go through one or more intermediate switches, wires that are required to go off-chip or off-board, communication overheads, and/or by the method which is used to prepare a packet for entry into the network. Link delays are an even greater concern for distributed machines and NOWs. This is because some latencies can be very high (due to the

fact that some processors can be far apart physically) and also because the variation among latencies can be high (since some processors may be very close or even part of the same tightly-coupled parallel machine).

Traditional Approaches

Since communication latency is an important factor in the performance of a parallel or distributed algorithm, several methods have been devised in an attempt to compensate for latency. The simplest of these methods is to slow down the computation to the point where the latency is accommodated. This approach is most commonly used at the circuit level, where the clock speed is set to be slow enough so that all of the data has time to reach its destination before the next step begins. This means that the circuit needs to be slowed down to accommodate the highest latency. Such an approach is clearly less than desirable in the context of a NOW with high-latency links.

An alternative approach is to organize the network in a hierarchical fashion so that the latencies are consistent with the hierarchy. For example, the CM-5 [1, 36] is organized into a fat tree and the KSR consists of two levels of nested rings. In both cases, the highest latency links are segregated into the top levels of the network hierarchy. This type of architecture works well for applications in which most of the computation is local since local computation can proceed using the low-level low-latency links. Only rarely, it is hoped, would the high latency links be needed. Thus, only certain steps of the computation would be slow. Unfortunately, this approach is not suitable for scenarios where the network is unstructured (which is often the case for a NOW) or when the underlying application requires frequent communications through the high-level links.

Redundant computation is another approach that has been used in the past [12, 30, 35] to hide the effects of latency. Here the idea is to avoid latency by recomputing data locally instead of waiting to receive it through a high-latency link.

Probably the most generally applicable method of hiding latency is the approach known as *complementary slackness.* The idea behind this approach is to load each

processor with enough work so that it stays productive while waiting for data to be supplied by the network. There are many implementations and incarnations of this method. For example, each processor in the CRAY YMP C-90 keeps busy by operating on a pipeline of 128 64-bit words. Processors on the HEP machine [53] swapped between unrelated threads while waiting for the data. The CM-1 and CM-2 were designed to simulate much larger virtual machines so that a single processor would perform the computation of many virtual processors [8, 57]. The technique also forms a critical component of Valiant's bulk synchronous model of parallel computing [59, 60] and it has been employed in several algorithms papers [5, 26, 30, 37, 50].

Unfortunately, in all of the preceding examples, it is incumbent on the programmer to provide the slackness or pipelining needed or to determine what part of the computation must be redundantly duplicated and by which processors to overcome the latencies in the network. Even in the scenario where a large virtual network is being simulated on a small parallel machine, it is incumbent on the programmer to find the parallelism necessary to efficiently implement the algorithm on a (potentially very large) virtual network.

Our Goal and Results

The goal of our research is to devise *automatic* methods for hiding latency. Our approach falls within the broad class of methods based on complementary slackness, but does not require the programmer to provide slackness, pipelines, or greater parallelism in order to hide the latency. Rather, our methods attempt to find the slackness automatically. By automatically finding the slackness, we hope to allow the programmer to assume that there are uniform delays on each link of the network, thereby easing the task of writing code. Moreover, our methods will enable us to automatically convert a program that was written for a well-structured unit-delay machine into a program that will run with minimal degradation in performance on a network with potentially large and variable latencies, at least for certain classes of networks.

In this thesis, we devise automatic methods for latency hiding on rings, linear arrays and 2-dimensional arrays. These are the simplest networks for parallel algo-

rithms. Nevertheless, they support a rich class of interesting and important parallel algorithms. A large number of examples can be found in [34]. We consider the problem of implementing any "dataflow" type of algorithm that is designed for an n-processor guest ring with unit-delay links on an n-processor host ring with arbitrary link delays. *A priori,* it would seem that any such implementation would require slowdown $d_{\max}$ where $d_{\max}$ is the largest delay in the host. Indeed, this is the delay that would be incurred by many prior approaches. Among other things, we show how to accomplish a slowdown of $O(\sqrt{d_{\text{ave}}})$, where d_{ave} is the average delay. Efficiency is preserved here if we use only $n/\sqrt{d_{\text{ave}}}$ processors to carry out the computation. The improvement is particularly impressive in the case when $d_{\max} \gg d_{\text{ave}}$, which is often the case for NOWs. We also consider the problem of simulating an $n \times n$ array with unit-delay links on an $n \times n$ array with arbitrary delays, and achieve a slowdown of $\tilde{O}(d_{\text{ave}}^{3/2})$. When large databases are involved in the computation we use the method of redundant computation for latency hiding. We postpone the detailed description of the computation models and our results to Chapter 2.

1.2 Dynamic Packet Routing with Delay Guarantees

In Part II of this thesis we study scheduling disciplines for packet-switched networks. Modern integrated services networks, such as broadband integrated services digital networks (B-ISDN), carry a wide range of traffic types over a single communication network. Traffic streams with different characteristics have different requirements. For example, file transfer requires absolute accuracy, but it can tolerate relatively high end-to-end delay, i.e. the transfer time. On the other hand, for real-time services such as audio, video and interactive multi-player games, it is crucial to provide quality-of-service (QoS) guarantees such as minimum delay and jitter.

The choice of an appropriate *scheduling discipline* is key to achieving performance bounds. Mitra and Ziedins [39] note that a good scheduling scheme satisfies both

fairness and *efficiency* constraints. The former guarantees a given grade of service to each application while the latter ensures the network resource is not underused. Keshav points out some common performance parameters in [28], including bandwidth, end-to-end delay, delay-jitter and loss. A *bandwidth bound* requires that each network user receives a minimum bandwidth from the network. End-to-end delay (or delay for short) is the total time from the packet injection until it is delivered to the desired destination. A *delay bound* requires fast delivery of the packets, which is particularly important for real-time services. A *delay-jitter bound* requires that the difference between the largest and the smallest delay is small. Finally, a *loss bound* requires that not many packets are lost. In practice, most current integrated-service networks provide only a bandwidth bound.

We concentrate on bounding the delay with no packet loss for connection-oriented networks, e.g. Asynchronous Transfer Mode networks (ATM). By connection oriented, we mean that a set of connections is predefined on the network, where each connection is specified by a route connecting a source node and a destination node. A user requests a share of a particular connection and injects a stream of packets along this connection continually.[1] When traveling from its source towards its destination, a packet goes through a set of switches along its predetermined route. More than one packet may contend for the same switch simultaneously. Due to limited processing power, some packets may have to queue up at the switches while others are being serviced. In this way, packets experience end-to-end delay. Apart from delay bounds, we are also concerned with queue size at each switch due to limited buffer size.

In order to bound the delay and the queue size, it is necessary to impose certain usage restriction on the users. We can view the relationship between the users and the scheduler as a contract. As long as the users generate the traffic within the agreed-upon rates, the scheduler can in return meet the users' performance requirements, e.g. providing the delay guarantee. *Leaky-bucket regulated traffic* is one popular method to

[1]We note the following. First, choosing routes for network connections is not a problem that we consider here. We focus on scheduling the motion of the packets once the routes are fixed. Second, we refer to our routing problem as *dynamic* since packets are injected over time. In contrast, in a *static* routing problem all packets are present initially.

restrict the users [14, 15, 58]. Here, a traffic stream is characterized by two parameters, a maximum burst size (or bucket size) and an average arrival rate. (Some papers characterize the traffic with one additional parameter, the maximum arrival rate.) Leaky-bucket regulated traffic is widely used in the literature, e.g. [18, 38, 48, 46, 47].

Related Work

The design of scheduling disciplines so as to minimize the end-to-end delay is well studied. In this section we discuss some related work.

GPS and GPS-Based Schemes One simple scheme is called *Generalized Processor Sharing (GPS)*, a generalization of the uniform *Processor Sharing (PS)* [29]. With PS, there is a separate FIFO queue for each contending user at a switch. During any time interval, a switch serves all the nonempty queues simultaneously at the same rate. This means that packets are serviced in infinitesimally small amounts by PS, i.e. it assumes a fluid model. GPS allows different users to have different service shares. A switch serves its nonempty queues in proportion to the service shares of the corresponding users. Since GPS uses an idealized fluid model that cannot be realized in the real world, various packet approximation algorithms of GPS have been proposed.

One simple emulation of GPS, proposed by Nagle, is called *Weighted Round Robin* [40]. In this approach, each of the nonempty queues is serviced in a round robin fashion proportional to the weight of the queue. Each time a queue is serviced a whole packet at the head of the queue is transmitted. Related work on round robin includes [23, 52].

Weighted Fair Queueing (WFQ), first proposed by Demers, Shenker and Keshav, is one of the best-known approximation schemes of GPS [17]. The intuition behind WFQ is to compute the times packets would complete service had GPS been used, and then serve packets in the order of these finishing times. Demers, et al. demonstrate the fairness of WFQ in [17]. A few congestion control algorithms are designed based on WFQ, e.g. [27, 51]. Some variants of WFQ are also studied. For example, *Worst-case*

Fair Weighted Fair Queueing by Bennett and Zhang [7], *Self-Clocked Fair Queueing* by Davin and Heybey [16] and Golestani [19].

Packet-by-packet Generalized Processor Sharing (PGPS), identical to WFQ, was proposed independently by Parekh and Gallager [46, 47]. They provide delay bounds for leaky-bucket regulated traffic streams. The first paper [46] proves that the delay bound of PGPS is within one packet transmission time of the bound of GPS *on a single switch.* The second paper [47] establishes end-to-end delay bounds for the case of multiple switches. In particular, if the ith connection has d_i switches and a packet arrival rate of r_i, then each packet injected along this connection can be delivered to its destination within $2d_i/r_i$ steps. These results are among the best known on the subject. (See Chapter 6 for more details.)

Other related disciplines include the *Virtual Clock Multiplexing* by Zhang [61], *Frame-based Fair Queueing* by Stiliadis and Varma [55, 56] and *Stop-and-Go Queueing* by Golestani [20, 21, 22].

Delay-Insertion-Based Schemes An entirely different technique based on "delay-insertion" is also used to bound the end-to-end delay. The intuition here is that if each packet receives a large random delay initially, then the packets are sufficiently spread out so that they only need to wait a small number of steps at each successive switch. Rabani and Tardos [49] and Ostrovsky and Rabani [41] prove delay bounds that are essentially the sum of the minimum injection rate and the maximum number of switches per connection. Their schemes also allow a packet loss probability. (See Chapter 6 for more details.)

Many techniques for analyzing delay-insertion-based schemes are introduced by Leighton et al. in [31, 32] in the context of static routing, where all the packets are present in the network initially. We shall summarize some of these techniques in Chapter 8.

Our Results

In this thesis we show, for the first time, how to design a schedule that guarantees asymptotically-optimal delay bounds for all connections. In particular, if the ith connection has d_i switches and a packet arrival rate of r_i, then each packet injected along this connection can be delivered to its destination in $O(d_i + 1/r_i)$ steps. Our result presents two main improvements upon previous work. First, the multiplicative bound of Parekh et al. is enhanced to an optimal additive bound. Second, the bound expressed in terms of $1/r_{\min} + d_{\max}$ due to Rabani et al. is enhanced to a connection-based delay guarantee of $O(1/r_i + d_i)$ for each connection i. An additional bonus of our work is small queues for all edges. We defer the detailed description of our model and results to Chapter 6.

Part I

Hiding Latency

Chapter 2

Overview

2.1 Model and Problem

We consider the problem of simulating a network G with unit-delay links on a network H with arbitrary delays on its links. We refer to G as the *guest* and H as the *host*. Let $g_1, g_2, \ldots$ be the processors of G and $p_1, p_2, \ldots$ be the processors of H. We shall use *pebbles* to record the computations performed by the guest processors. In particular, pebble (i,t) represents the tth step of computation by processor g_i. In a *simulation* of G, H carries out the same step-by-step computation as G. In other words, H simulates G by computing every pebble created by G in an order that preserves the "dependency" of the pebbles. Our goal is to provide methods that would allow H to simulate G with a minimum amount of slowdown when G is used in a general purpose way. Two computation models are studied here, the *dataflow model* and the *database model.*

Dataflow Model

In the dataflow model, each computation solely depends on the computation of the previous step. Creating a pebble (i,t) involves two time units. The first time unit is for communication, where g_i obtains pebbles of the form $(j,t-1)$ from all its neighbors g_j. The second time unit is for computation, where g_i performs computation based

on pebbles $(j, t-1)$ and records the result in pebble (i, t). Take an example of an n-node guest linear array. In $2T$ time steps, G creates $n \times T$ pebbles, where pebble (i, t), for $1 < i < n$ and $1 < t \leq T$, depends on pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$. (See Figure 2-1.) Any host processor p can compute pebble (i, t) as long as p has the information in pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$ either by directly computing these pebbles or by receiving them from neighboring processors.

The dataflow model is applicable to many computations such as matrix operations, Fourier transform, sorting, algorithms for computational geometry, etc. A large number of examples can be found in [34].

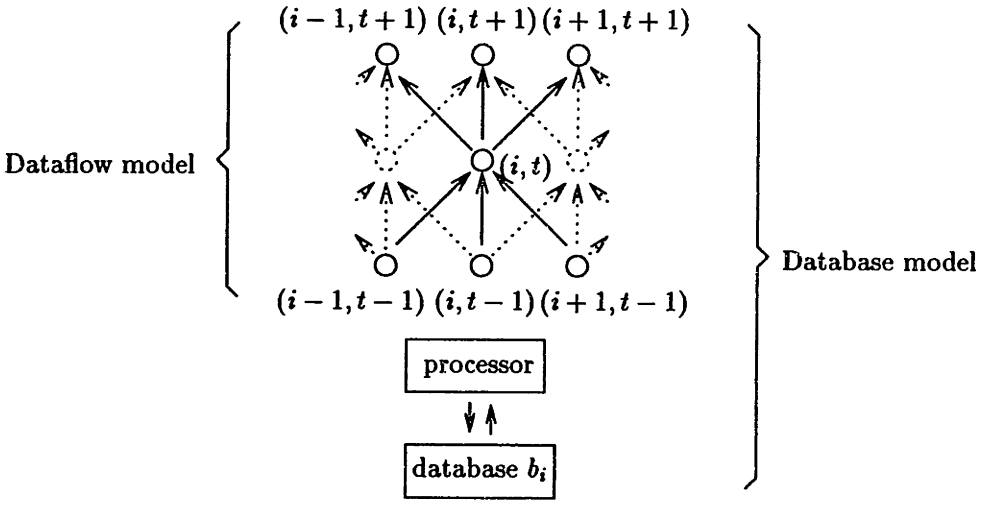


Figure 2-1: The computation pebbles created by a guest linear array.

Database Model

In the database model each guest processor g_i has a potentially large local memory that may be accessed and updated by g_i during each step. We refer to the local memory of g_i as the *database*, b_i. Each computation not only depends on the computation of the immediate past but also the state of the database. For example, let G be a linear array. To create pebble (i, t), g_i first communicates with its neighbors, then performs computations based on pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$ and the current state of database b_i. Lastly, g_i updates database b_i. Hence, creating a pebble involves two time units as in the dataflow model, one for communication and

one for computation and recording.

In the database model, a pebble not only records the result of a computation but also the *changes* to the database incurred by this computation. To emphasize, a pebble does not contain a snapshot of the whole database but rather the changes incurred by one computation. Therefore, a pebble has small size and can be passed along links.

In order to simulate G on H, we assume that the initial contents of each database can be copied *before* the computation begins (thereby allowing redundant computations), but that the large size of a database makes it impractical to transmit a copy of a database through the network *during* the computation. Suppose processor p of H copies databases b_i and b_j, then p only has access to b_i and b_j and hence can only compute pebbles of the form (i,t) and (j,t) for $t \geq 1$. Moreover, if both processors p and q decide to copy b_i, then p and q each maintains a copy of b_i, and each looks up and updates its own copy. If p is to compute pebble (i,t) then p needs an updated copy of the database that includes all the changes incurred by the computations (i,t') for all $t' < t$. Hence, p must either have directly computed all the pebbles (i,t') or else have received the information from its neighbors.

Unlike the dataflow model, the database model captures a scenario where the computation performed by a processor depends on the state of a local memory or where part of the computation performed by a processor is to update its local memory. These situations could be critical in some applications involving a network of workstations.

Bandwidth

The guest network G has unit bandwidth on each link. This allows each pebble to be passed along a unit-delay link of G in one time step. In our simulation we assume that the link bandwidth of the host network H is w. That is, P pebbles can be passed along a d-delay link of H in $d + \lceil \frac{P}{w} \rceil - 1$ steps by pipelining. In many cases of our study, it is sufficient to assume that the host and the guest have comparable link bandwidth, i.e. w is a constant. However, in certain situations the bandwidth needs

to be $\tilde{O}(\log n)$. Otherwise, we pay an extra factor of $\tilde{O}(\log n)$ in the slowdown. The details are discussed in Sections 4.2.4 and 5.2.4.

2.2 Results

Table 1 summarizes our results. In the table, n is the size of the guest, d_{ave} is the average delay of the host and "Bd-deg" stands for bounded-degree. The ratio of n and the slowdown is the size of the host since all the simulations are *work efficient*, i.e. it takes the guest and the host the same amount of work to compute the same result, where *work* is the product of the number of processors used and the running time.

	Guest	Host	Model	Order of Slowdown
1	Ring/Linear Array	Bd-deg Network	Dataflow	$\sqrt{d_{\text{ave}}}$
2	Ring/Linear Array	Bd-deg Network	Database	$\sqrt{d_{\text{ave}}}\log^3 n$
3	2-D Array	2-D Array	Dataflow	$d_{\text{ave}}^{2/3}\log^{5/3} n$
4	2-D Array	Bd-deg Network	Dataflow	$n^{1/4}(\sqrt{d_{\text{ave}}}+n^{1/4})$
5	2-D Array	Bd-deg Network	Database	$n^{1/4}\log^3 n(\sqrt{d_{\text{ave}}}+n^{1/4})$

Table 1: Result Summary.

The first two results in Table 1 are proved in terms of linear arrays. An n-node unit-delay ring is essentially the same as an n-node unit-delay linear array, since the latter can simulate the former with a slowdown of 2 [34]. Result 1 is asymptotically optimal in some cases. In addition, we also have a constant-approximation algorithm for simulating rings and linear arrays in the dataflow model. Results 2 and 3 are optimal up to a polylogarithmic factor in some cases. Result 3 is for a worst-case model. When the delays on the host are randomly arranged, the bound can be improved to $O(d_{\text{ave}}^{2/3})$. Results 4 and 5 are easy generalizations of Results 1 and 2 respectively. Chapters 3 and 4 present latency hiding methods for the dataflow model. Chapter 5 concentrates on the database model.

The methods for latency hiding in the two computation models are substantially different. For example, we make heavy use of redundant computation in the database model, whereas redundancy is apparently not useful for the dataflow model.

Our bounds indicate that hiding latency in the database model is more difficult than in the dataflow model. Intuitively, this is because computation in the dataflow model is processor independent, and hence can be done by any processor with the information of the previous computation. In the database model computation can only be done by the processors with the right databases. One cannot afford to pass large databases across the links with limited bandwidth, because this will cause high slowdown. One also cannot afford to keep many copies of the databases, because memory is expensive and keeping every copy of the databases updated is difficult.

In Chapter 5, we also establish limits on the degree to which the high latency can be mitigated when each database is allowed a small number of copies. For example, if each database has only one copy, we show that the slowdown can be as much as $d_{\max}$ even if d_{ave} is a constant and the best simulation is used. When each database has at most two copies and each host processor copies a constant number of databases, we give an example of a host whose average delay is a constant, but for which the slowdown has a lower bound of $\Omega(\log n)$. These results demonstrate that it is easier to overcome latencies in dataflow types of computations than in computations that require access to large local databases.

2.3 A Related Scheduling Problem

The problem of latency hiding in the dataflow model can be viewed as the following scheduling problem. The pebbles created by the guest network together with their dependencies form a directed acyclic graph (dag), whose nodes represent computational tasks of equal execution time, and whose arcs represent precedence. All these tasks are to be computed by the processors in a given host network. If the same host processor computes two tasks of direct dependence, no communication cost is incurred. Otherwise, there is a communication cost between the two host processors

that compute these two tasks, and this cost is equal to the total delay between the processors in the host network. The goal here is to schedule the dag (with possible repetitions of the nodes) using the given host processors so as to minimize the *makespan*, i.e. the total time taken to execute all the tasks.

A variation of the above scheduling problem has been studied. Here, we are given any task dag (not necessarily created by a guest network in the dataflow model). All the arcs in the dag are associated with a fixed quantity that indicates the communication cost. Note that, unlike our problem, the communication cost here is the same for any processor-pair. In [44] Papadimitriou and Ullman studied an $n \times n$ grid dag (which they called a diamond dag). They showed a nontrivial time-communication tradeoff and gave an asymptotically-optimal schedule. Their result was similar to the special case of our Result 1 stated in Section 2.2 where all the link delays in our host network are the same. In [45] Papadimitriou and Yannakakis presented a 2-approximation algorithm for general dags where an unlimited number of processors could be used. For well-known families of dags such as the full binary tree, the diamond dag and the fast Fourier transform, only a finite number of processors were needed and their approximation algorithms were optimal (or near-optimal). Redundant computation was used in [45].

Dag scheduling has been studied in other papers, including [2, 11, 13, 24, 25, 42, 43]. Some variations of the problem include the cases in which the dags are limited to certain topologies, the task nodes require different execution time, arcs require different communication times and/or processors have different processing powers.

Chapter 3

Dataflow Model – Linear Arrays

We begin our presentation with the methods for hiding latency in linear arrays. Our basic approach is to transfer a process that involves a two-way communication to a process that involves one-way communication only. (This idea is also essential for simulating 2-dimensional arrays in Chapter 4.) We present an asymptotically tight bound on the slowdown for linear arrays. All the results for linear arrays are applicable to rings.

3.1 Average Delay – An Upper Bound

Let the network G be an n-processor guest linear array with unit delay on all the edges. Let the network H be an n-processor host linear array with arbitrary delays, where d_i is the delay on the ith edge of H. As discussed in Section 2.1, in $2T$ time steps G creates $n \times T$ pebbles, where pebble (i,t), for $1 < i < n$ and $1 < t \leq T$, depends on pebbles $(i-1,t-1)$, $(i,t-1)$ and $(i+1,t-1)$. We first present algorithm STRIPE in which H simulates G with a slowdown of $O(d_{\text{ave}})$, where $d_{\text{ave}} = \sum_{k=1}^{n-1} d_k/(n-1)$ is the average delay of H.

Consider the first $n/2$ rows of pebbles created by G. Let L be the triangle formed by pebbles (i,t), where $i+t \leq n+1$. Let R be the triangle formed by pebbles (i,t), where $i \leq t$. (See Figure 3-1.) In STRIPE, H first simulates the bottom half of L and then the bottom half of R. At this point every pebble in the first $n/2$ rows is

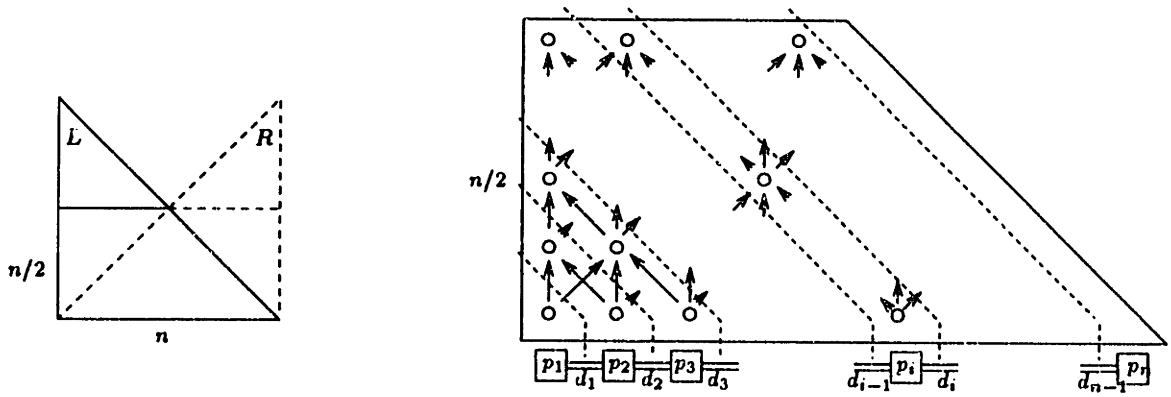


Figure 3-1: (Left) Triangles L and R. (Right) Algorithm STRIPE. Each slanted stripe is simulated by one processor of H. Arrows correspond to communications. Dashed lines correspond to the delays d_i encountered by communications.

simulated. If the entire computation of G is partitioned into groups each of which consists of $n/2$ rows of pebbles, then H can repeat the process and simulate every group in a similar manner.

To simulate the bottom half of L, the computation pebbles of G are divided into n slanted stripes, and each processor of H simulates one stripe. (See Figure 3-1.) In particular, processor p_i of H simulates a stripe consisting of pebbles $(i-t+1,t)$, for $1 \leq t \leq i$ and $t \leq n/2$. Note that in the original computation by G, processor g_i depends on both g_{i-1} and g_{i+1}. However, in the simulation by H p_i depends on p_{i-1} and p_{i-2}. Hence, STRIPE transforms a process that involves two-way communication into a process that involves only one-way communication.

Lemma 3.1.1 *Processor p_i $(1 \leq i \leq n)$ is able to compute pebble $(i-t+1,t)$ at step $t + \sum_{k=1}^{i-1} d_k$.*

Proof: We use induction on i. The base case for p_1 is obvious. Pebble $(i-t+1,t)$ depends on pebbles $(i-t,t-1)$, $(i-t+1,t-1)$ and $(i-t+2,t-1)$, which are computed by processors p_{i-2}, p_{i-1} and p_i respectively. By induction these three pebbles are computed at step $(t-1)+\sum_{k=1}^{i-3} d_k$, $(t-1)+\sum_{k=1}^{i-2} d_k$ and $(t-1)+\sum_{k=1}^{i-1} d_k$ respectively. It follows that $(i-t+1,t)$ can be computed at step $t+\sum_{k=1}^{i-1} d_k$. □

Hence, pebbles $(i+1,n/2)$, for $0 \leq i \leq n/2$, are computed at steps $n/2 + \sum_{k=1}^{i+n/2-1} d_k$, and so the bottom half of L is simulated in $n/2 + \sum_{k=1}^{n-1} d_k$ steps by

H. The bottom half of R is simulated in a similar manner after processor p_n passes pebbles $(n-t+1,t)$, for $1 \leq t \leq n/2$, to appropriate processors. Note that the intersection of R and L is only computed once. Thus, H has completed simulating the first $n/2$ rows of pebbles created by G. The next $n/2$ and every subsequent $n/2$ rows of pebbles can be simulated in a similar manner. Therefore, the slowdown is upper bounded by,

$$s = 2\frac{n/2 + \sum_{k=1}^{n-1} d_k}{n} = O(d_{\text{ave}}).$$

3.2 A Better Upper Bound

To get a better upper bound on the best achievable slowdown, we use the idea of "complementary slackness" in our new algorithm called FATSTRIPE. Each host processor is loaded with enough work to balance out the communication time. Suppose FATSTRIPE uses an interval of m processors to carry out the simulation. For simplicity, assume that this interval consists of processors $p_1, \ldots, p_m$. The bottom half of L is divided into m slanted stripes, each of which has width $\ell = n/m$. Again, p_i computes every pebble in stripe i. (See Figure 3-2.) Within each stripe i, p_i first computes all the pebbles in the bottom row and then moves up.

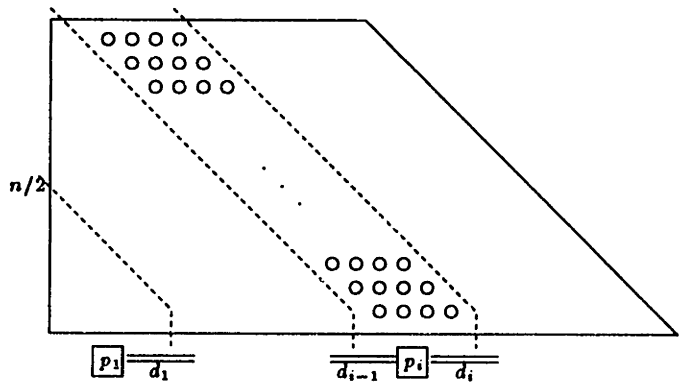


Figure 3-2: Algorithm FATSTRIPE. Processor p_i simulates stripe i which has width $\ell = n/m$. (In the figure, $\ell = 4$.) All the pebbles in stripe i are computed by time step $\ell n/2 + \sum_{k=1}^{i-1} d_k$.

Lemma 3.2.1 *Processor p_i finishes simulating stripe i by step $\ell n/2 + \sum_{k=1}^{i-1} d_k$.*

Proof: We inductively show that p_i can compute the pebbles in the xth row of stripe i by time step $\ell x + \sum_{k=1}^{i-1} d_k$. The base of the induction holds trivially for $i = 1$ and $x = 1$, since processor p_1 does not depend on other processors and pebbles in the first row do not depend on other pebbles. Let us consider the pebbles on the $(x+1)$st row of stripes $i+1$, for $x \geq 1$ and $i \geq 1$. These pebbles could only depend on pebbles on the xth row of stripe $i-1$, i and $i+1$, which can be computed by processors p_{i-2}, p_{i-1} and p_i by steps $\ell x + \sum_{k=1}^{i-3} d_k$, $\sum_{k=1}^{i-2} d_k$ and $\ell x + \sum_{k=1}^{i-1} d_k$ respectively by induction. Hence, p_i is able to receive all the information necessary to compute its $(x+1)$st row by step $\ell x + \sum_{k=1}^{i-1} d_k$ and therefore finish computing the $(x+1)$st row by step $\ell(x+1) + \sum_{k=1}^{i-1} d_k$. Since each stripe contains at most $n/2$ rows, p_i finishes simulating stripe i by step $\ell n/2 + \sum_{k=1}^{i-1} d_k$. □

Hence, the slowdown is $O(n/m + \sum_{k=1}^{m-1} d_k/n)$ in simulating the first $n/2$ rows of pebbles. All the subsequent $n/2$ rows can be simulated in a similar manner. To minimize the slowdown, FATSTRIPE uses the interval I (with m_I processors and d_I average delay) that minimizes the quantity $n/m_I + d_I m_I/n$. Therefore,

Theorem 3.2.2 FATSTRIPE *achieves a slowdown of* $\min_{\text{intervals } I} O(n/m_I + d_I m_I/n)$.

In the case when $\sqrt{d_{\text{ave}}} \leq n$, there exists an interval I with $M_I = n/\sqrt{d_{\text{ave}}}$ processors and average delay $d_I \leq d_{\text{ave}}$. Theorem 3.2.2 implies that the slowdown is $O(\sqrt{d_{\text{ave}}})$ when M_I simulates G. In the case when $\sqrt{d_{\text{ave}}} > n$ a single host processor is used to carry out the simulation, which incurs a slowdown of $n = O(\sqrt{d_{\text{ave}}})$. The simulation is work-efficient in both cases. Therefore,

Corollary 3.2.3 FATSTRIPE *efficiently simulates G on H and achieves a slowdown of $O(\sqrt{d_{\text{ave}}})$, where d_{ave} is the average delay of H.*

Bandwidth Let us consider the effect of bandwidth on the slowdown. In FATSTRIPE as long as the stripe width is at least 2, then pebbles cross the edges one at a time by using pipelining. In STRIPE (i.e. FATSTRIPE with stripe width 1) at most two pebbles may cross an edge at the same time. Therefore, it is sufficient for the host bandwidth to be twice as large as that of the guest bandwidth. Otherwise, we pay another factor of 2 in the slowdown.

3.3 A Matching Lower Bound

We proceed to show that the upper bound, $\min_I O(n/m_I + d_I m_I/n)$, in Theorem 3.2.2 is asymptotically tight by showing that $\min_I \max\{n/2m_I, d_I m_I/2n\}$ is a lower bound on the best achievable slowdown even if we allow *redundant computation.* Note that with redundant computation, a pebble may be computed by several host processors. This technique makes it more likely for the host to simulate the guest efficiently. However, we show below that redundancy does not help in this case.

Lemma 3.3.1 *The top pebble,* $(1,n)$ *of triangle* L*, cannot be computed at a time step earlier than*

$$\tau = \min_{\text{intervals } I} \max\{n^2/2m_I, d_I m_I/2\}.$$

Proof: We consider how the pebbles in L are computed in some simulation of G by H. In particular, we build a ternary tree T to keep track of the processors that have "effectively" computed the pebbles in L. The top pebble $(1,n)$ has to be computed by some processor of H. Call this processor q. (If more than one processor of H has computed $(1,n)$, then we pick any one of them to be q.) We label the root of tree T with $q^{(1,n)}$. Let u be a processor that has computed $(1,n-1)$ *and* has passed this information to q, and v be a processor that has computed $(2,n-1)$ *and* has passed this information to q. (Note that other processors may compute $(1,n-1)$ and $(2,n-1)$. We are only concerned with processors that pass information to q.) Now label the children of $q^{(1,n)}$ with $u^{(1,n-1)}$ and $v^{(2,n-1)}$. We proceed to construct the children of $u^{(1,n-1)}$ and $v^{(2,n-1)}$. In general, node $a^{(i,t)}$ in T has children $b^{(i-1,t-1)}$, $c^{(i,t-1)}$ and $d^{(i+1,t-1)}$ if the following holds. Processors a, b, c and d compute pebbles (i,t), $(i-1,t-1)$, $(i,t-1)$ and $(i+1,t-1)$ respectively, *and* a receives the values of $(i-1,t-1)$, $(i,t-1)$ and $(i+1,t-1)$ from b, c and d before a is able to compute (i,t). The leaves of T are nodes of the form $p^{(i,1)}$. The important observation is the following. If $p^{(i,t)}$ is a node in T, then information has to be passed from processor p to q in H. The total delay from p to q lower bounds the number of steps in the simulation.

Let J be the smallest interval that contains all the processors appearing in tree T. If processors x and y are at the two ends of J, then there exist two nodes of the form $x^{(i_x,t_x)}$ and $y^{(i_y,t_y)}$ in T. Hence, information has to be passed from x and y to q in H. This takes at least $d_J m_J/2$ steps, and pebble $(1,n)$ therefore cannot be computed at a step earlier than $d_J m_J/2$. By a work argument, $(1,n)$ cannot be computed before step $n^2/2m_J$. Hence, $(1,n)$ cannot be computed at a step earlier than $\tau = \min_I \max\{n^2/2m_I, d_I m_I/2\}$. □

It follows that the slowdown in simulating triangle L is lower bounded by τ/n. By a similar argument to Lemma 3.3.1 none of the pebbles (i,n), for $1 \leq i \leq n$, can be computed at a time step earlier than τ. By repeating this argument the first kn rows of G cannot be simulated in time less than $k\tau$. Therefore, we obtain,

Theorem 3.3.2 *The slowdown of any simulation of an n-node guest linear array G by a host linear array H is lower bounded by $\min_I \Theta(n/m_I + d_I m_I/n)$, where I is a subarray of H and has m_I processors and average delay d_I. Hence,* FATSTRIPE *is optimal up to a constant factor.*

3.4 Simulating Linear Arrays on General Networks

We now consider simulating a linear array G on a general n-node network H with average delay d_{ave}. We first embed a linear array $\mathcal{H}$ in H and then use $\mathcal{H}$ to carry out the simulation of G.

Lemma 3.4.1 *Let H be a connected n-node network with arbitrary topology. Then an n-node linear array $\mathcal{H}$ can be one-to-one embedded in H such that every edge of H is used at most twice in $\mathcal{H}$.*

Proof: Our proof follows the approach of Theorem 3.15 in [34, page 470]. It is sufficient to embed a linear array $\mathcal{H}$ in a spanning tree of H. The proof proceeds by induction on the height of the tree with the following inductive hypothesis. For any child u of the root v, there is a one-to-one embedding of a linear array in the tree such that v and u form two endpoints of the array, the edge uv is used at most once

and all other edges of the tree are used at most twice. (Note that we treat all the edges as undirected.)

Let T be any spanning tree of H. The base of the induction in which T is a single node, i.e. the height is 0, is trivial. Otherwise, let v be the root of T and u be any child of v. We label the children of v as $u_1, \ldots, u_d$, and assume $u = u_d$ without loss of generality. We place the first node of the linear array at v, and place the second node of the array at any child w of u_1 (if any) using edges vu_1 and u_1w. Next, we inductively place the nodes of the array in each node of the subtree of T rooted at u_1, making sure that the last node is placed at u_1, the edge u_1w is used at most once and that all other edges in the subtree are used twice. Therefore, edge u_1w is used at most twice in total.

We place the next node of the linear array at any child x of u_2 (if any), using edges u_1v, vu_2 and u_2x. Again, we inductively place nodes of the linear array in the subtree rooted at u_2 such that u_2 and x are endpoints. We continue in this fashion. At the last subtree rooted at u, we enter this subtree at a child of u (if any) and exit at u. This completes the embedding of the linear array. Our lemma follows from the observation that the linear array has endpoints v and u, edges vu_1, ..., vu_{d-1} are used twice and vu_d is used once. (See Figure 3-3.)

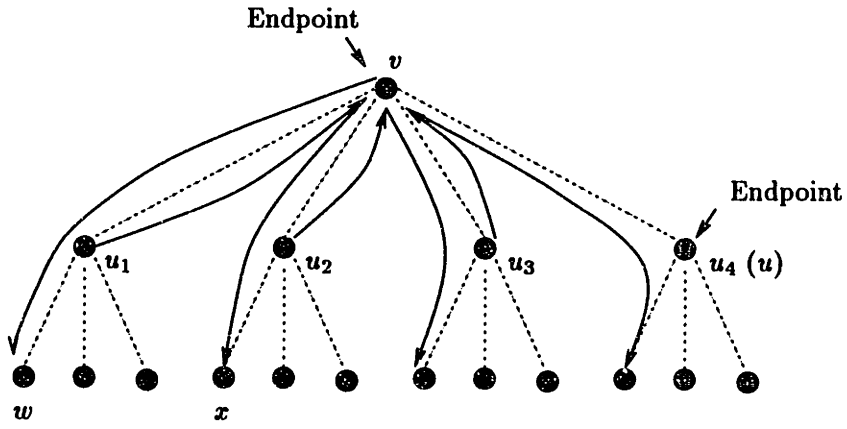


Figure 3-3: Embed a linear array one-to-one in a tree such that each tree edge is used at most twice. The dotted lines indicate tree edges and the solid lines indicate array edges.

□

Since H has n nodes and degree δ, H has at most $\delta n/2$ edges and therefore the total delays on all edges of H is at most $\delta d_{\text{ave}} n/2$. By Lemma 3.4.1, $\mathcal{H}$ uses each edge of H at most twice. Hence, the total delays on all edges of $\mathcal{H}$ is at most $\delta d_{\text{ave}} n$, and the average delay of $\mathcal{H}$ is at most δd_{ave}. By Corollary 3.2.3, $\mathcal{H}$ can simulate G with a slowdown of $O(\sqrt{\delta d_{\text{ave}}})$. When H has bounded degree, i.e. $\delta = O(1)$, we have,

Theorem 3.4.2 *A bounded-degree host network with average delay d_{ave} can efficiently simulate an n-processor guest linear array with a slowdown of $O(\sqrt{d_{\text{ave}}})$.*

Theorem 3.4.2 does not hold when H has unbounded degree. Consider the following example. Let H be a linear array of $\sqrt{n}$ cliques, in which each clique contains $\sqrt{n}$ nodes. If a clique edge has delay 1 and an edge connecting two adjacent cliques has delay n, then H has $d_{\text{ave}} < 4$. Suppose m connected cliques are used to simulate n steps of G. Lemma 3.3.1 implies a slowdown of $\min_m \max\{\sqrt{n}/2m, m/2\}$ in simulating every n steps of computation by the guest. The first term follows from a work argument, since $m\sqrt{n}$ processors are in m cliques. The second term comes from the communication delay, since a linear array embedded in these m connected cliques has a total delay of at least mn. Hence, the slowdown is at least $\min_m \max\{\sqrt{n}/2m, m/2\}$, which is $\Omega(n^{1/4})$, whereas the average delay is a constant.

Chapter 4

Dataflow Model – Two-Dimensional Arrays

In this chapter we present methods for hiding latency in 2-dimensional arrays. The analysis here is substantially more complex than that for the 1-dimensional case. We focus on simulating a 2-dimensional array on a 2-dimensional array. Section 4.1 generalizes the approach for the linear arrays. Section 4.2 introduces some new mechanism to improve the bound. Section 4.3 discusses the case when the delays are randomly arranged.

4.1 An Analogue of the One-Dimensional Case

Let the guest network G be an $n \times n$ 2-dimensional array with unit delay on all the edges. Let the host network H be an $n \times n$ 2-dimensional array with arbitrary delays. Let $x_{i,j}$ be the delay between processors $p_{i,j}$ and $p_{i+1,j}$ of H for $1 \leq i \leq n-1$ and $1 \leq j \leq n$, and let $y_{i,j}$ be the delay between $p_{i,j}$ and $p_{i,j+1}$ of H for $1 \leq i \leq n$ and $1 \leq j \leq n-1$. The tth step of computation by processor $g_{i,j}$ of G is recorded in pebble (i,j,t). In $2T$ steps, G creates $n \times n \times T$ pebbles, where pebble (i,j,t), for $1 < i,j < n$ and $1 < t \leq T$, depends on $(i-t+1, j-t+1, t-1)$, $(i-t, j-t+1, t-1)$, $(i-t+2, j-t+1, t-1)$, $(i-t+1, j-t, t-1)$ and $(i-t+1, j-t+2, t-1)$.

Consider the first $n/2$ steps of computation by G. We define four pyramids P_1, P_2,

P_3 and P_4 analogous to the left and right triangles in the linear array case. All four pyramids have the square, defined by vertices $(1,1,1)$, $(1,n,1)$, $(n,1,1)$ and $(n,n,1)$, as their bases. The top vertices of P_1, P_2, P_3 and P_4 are $(1,1,n)$, $(1,n,n)$, $(n,1,n)$ and (n,n,n) respectively. Note that the bottom half of the four pyramids contain all the pebbles created by G for the first $n/2$ steps of computation.

Algorithm 2D-RAY is a 2-dimensional analogue of STRIPE. To simulate the first $n/2$ steps of computation of G, 2D-RAY simulates P_1, P_2, P_3 and P_4 one by one. Pyramid P_1 is divided into n^2 rays, each of which is simulated by one processor of H. In particular, processor $p_{i,j}$ of H simulates ray $R_{i,j}$, consisting of pebbles $(i-t+1, j-t+1, t)$, for $1 \leq t \leq \min\{i, j, n/2\}$. (See Figure 4-1.) When every pebble for the first $n/2$ steps of computation of G is simulated, 2D-RAY repeats the process and simulates the next $n/2$ steps of computation. In the following we bound the slowdown in terms of the total delay on *monotone paths*, where a monotone path travels in two directions, up and right. Let the length of a path be the total delay on the path, and let $D_{i,j}$ be the length of the longest monotone path from processor $p_{1,1}$ to $p_{i,j}$ in H. We have,

Lemma 4.1.1 *Processor $p_{i,j}$ of H is able to compute pebble $(i-t+1, j-t+1, t)$ at step $D_{i,j}+t$.*

Proof: We use induction on the indices (i,j) of the processors. The base of the induction for $p_{1,1}$ is obvious. Pebble $(i-t+1, j-t+1, t)$ depends on pebbles $(i-t+1, j-t+1, t-1)$, $(i-t, j-t+1, t-1)$, $(i-t+2, j-t+1, t-1)$, $(i-t+1, j-t, t-1)$ and $(i-t+1, j-t+2, t-1)$, which are computed by processors $p_{i-1,j-1}$, $p_{i-2,j-1}$, $p_{i,j-1}$, $p_{i-1,j-2}$ and $p_{i-1,j}$ respectively. (See Figure 4-1.) By induction, these five pebbles are computed at steps $D_{i-1,j-1}+(t-1)$, $D_{i-2,j-1}+(t-1)$, $D_{i,j-1}+(t-1)$, $D_{i-1,j-2}+(t-1)$ and $D_{i-1,j}+(t-1)$ respectively. It follows that pebble $(i-t+1, j-t+1, t)$ can be computed at step $\max\{D_{i-1,j}+x_{i-1,j}, D_{i,j-1}+y_{i,j-1}\}+t = D_{i,j}+t$.

□

Hence, 2D-RAY simulates pyramid P_1 in $D_{n,n}+n$ steps. Since P_2, P_3, and P_4 can be simulated similarly, 2D-RAY simulate the first $n/2$ steps of computation of G in

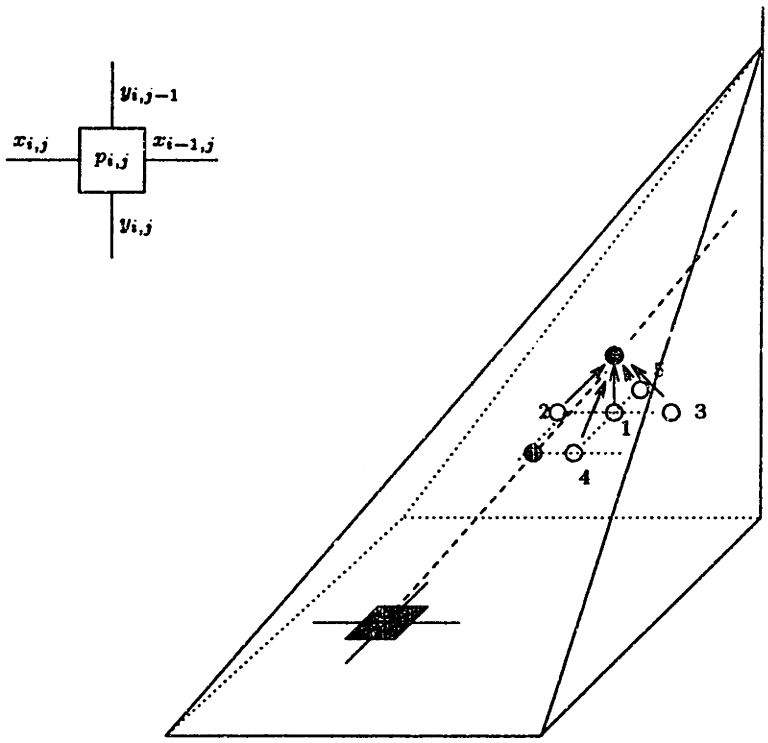


Figure 4-1: Algorithm 2D-RAY. In pyramid P_1 the dashed line represents the ray of pebbles computed by processor $p_{i,j}$ (which is shown in the upper left corner). Two of those pebbles, computed at times t and $t-1$, are shown shaded. The five numbered pebbles are those that $(i-t+1, j-t+1, t)$ depends on.

$O(D_{n,n}+n)$ steps. The simulation is repeated for every $n/2$ steps of computation of G. Therefore,

Lemma 4.1.2 *Algorithm* 2D-RAY *achieves a slowdown of* $O(D_{n,n}/n)$, *where* $D_{n,n}$ *is the length of the longest monotone path in* H.

Unfortunately, $D_{n,n}$ can be large compared with d_{ave}, the average delay of H. In the worst case $D_{n,n}$ can be $\Theta(n^2 d_{\text{ave}})$, implying a slowdown of $\Theta(n d_{\text{ave}})$. We introduce algorithm FATRAY, a two-dimensional analogue of FATSTRIPE, to achieve a slowdown that is often better than $O(D_{n,n}/n)$. Pyramid P_1 is divided into m^2 rays, each of which has size $\ell \times \ell = \frac{n}{m} \times \frac{n}{m}$. FATRAY uses an $m \times m$ contiguous subarray of processors in H to carry out the simulation. For simplicity, assume FATRAY uses processors $p_{i,j}$ $(1 \le i,j \le m)$. Again, $p_{i,j}$ computes every pebble in ray $R_{i,j}$, and $p_{i,j}$ first computes all the pebbles on the bottom plane and then moves up. The follow lemma is analogous to Lemma 3.2.1.

Lemma 4.1.3 *Processor $p_{i,j}$ finishes simulating ray $R_{i,j}$ by step $\ell^2 n/2 + D_{i,j}$.*

Proof: As in Lemma 3.2.1 we can inductively show that $p_{i,j}$ can compute all the pebbles in the xth plane in ray $R_{i,j}$ by time step $\ell^2 x + D_{i,j}$. Since each ray contains at most $n/2$ planes of pebbles, $p_{i,j}$ finishes simulating ray $R_{i,j}$ by step $\ell^2 n/2 + D_{i,j}$. □

This implies a slowdown of $O(n^2/m^2 + D_{m,m}/n)$. To minimize the slowdown, FATRAY uses the contiguous subarray S that minimizes $n^2/m_S^2 + D_S/n$, where $m_S \times m_S$ is the size of S and D_S is the length of the longest monotone path in S.

Theorem 4.1.4 FATRAY *achieves a slowdown of* $\min_{\text{subarrays } S} O(n^2/m_S^2 + D_S/n)$.

Unfortunately, the slowdown can still be big compared with d_{ave}. For example, suppose that H has n edges of delay n which are spread out evenly in the network and has unit delay on all other edges. The slowdown is $\min_S \Theta(n^2/m_S^2 + D_S/n) = \Theta(n^{1/3})$ whereas d_{ave} is a constant. Matters are better, however, when all the delays are the same, as we show in the following theorem.

Theorem 4.1.5 *In the case where all the delays in H are d,* FATRAY *efficiently simulates G on H and achieves a slowdown of $\Theta\left(\min\{d^{2/3}, n^2\}\right)$. The slowdown is optimal up to a constant factor.*

Proof: When $d \leq n^3$ FATRAY uses a subarray of size $\frac{n}{d^{1/3}} \times \frac{n}{d^{1/3}}$. Theorem 4.1.4 implies a slowdown of $O(d^{2/3})$. We show that the slowdown is asymptotically tight as follows. Consider pebble $(i, j, d^{1/3})$, and suppose processor q computes it in a simulation. Let A be the set the pebbles of the form (i', j', t), for $1 \leq t < d^{1/3}$, on which $(i, j, d^{1/3})$ depends, i.e. $(i, j, d^{1/3})$ cannot be computed until after (i', j', t) is computed. If every pebble in A is computed by q then it takes at least $|A| = \Omega\left((d^{1/3})^3\right) = \Omega(d)$ time steps to simulate A. Otherwise, a processor $p \neq q$ computes some pebble in A and passes this information to q. The delay from p to q is at least d. Hence, the slowdown on simulating the first $d^{1/3}$ steps is $d^{2/3}$. The same argument applies for the slowdown in the next $d^{1/3}$ steps.

When $d > n^3$ FATRAY uses a single host processor for the simulation and achieves a slowdown of $O(n^2)$. This slowdown is asymptotically tight for the same reason as

in the previous case. We consider pebbles (i, j, n) instead of $(i, j, d^{1/3})$. In both cases the simulation is work-efficient. □

Theorem 4.1.5 can be generalized to any k-dimensional array, for $k \geq 1$.

Theorem 4.1.6 *Suppose G is an $n \times \ldots \times n$ k-dimensional array with unit-delay edges, and H is an $n \times \ldots \times n$ k-dimensional array with delay-d edges, then H can efficiently simulate G with a slowdown of $\Theta(\min\{d^{k/k+1}, n^k\})$. The slowdown is optimal up to a constant factor.*

4.2 Improved Bounds for Worst-Case Delays

In order to improve the slowdown, we observe that not all the host processors are useful. If a host processor is surrounded by high delays, then the benefit to be gained by using its computing power is nullified by the communication cost. We first describe criteria of removing such host processors. We then embed guest processors to the unremoved host processors. Suppose that guest processor $g_{i,j}$ is mapped to host processor p, then p computes the pebbles in ray $R_{i,j}$ in the 2D-RAY algorithm. For any arrangement of the delays in H, we show how to embed G on H such that, for any monotone path in G, its image in H has length of $O(d_{\text{ave}} n \log^{5/2} n)$. As a result, Lemma 4.1.2 implies a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$ as long as only $O(1)$ guest processors are mapped to each host processor. By applying the idea used in FATRAY, we improve the slowdown to $O(d_{\text{ave}}^{2/3} \log^{5/3} n)$ and achieve work-efficiency at the same time.

4.2.1 Removing Useless Processors

We first recursively represent H using a quad-tree, in which each node corresponds to a subarray of H. The root represents the entire $n \times n$ array. The four children of the root represent the four $\frac{n}{2} \times \frac{n}{2}$ subarrays, etc. In general, a node at depth k of the quad-tree corresponds to an $\frac{n}{2^k} \times \frac{n}{2^k}$ subarray of H. We refer to this subarray as a *depth-k* array. The leaves represent the individual processors of H. (See Figure 4-2.)

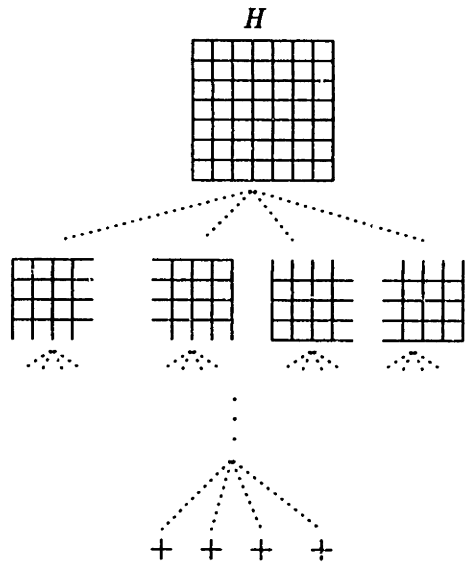


Figure 4-2: The quad-tree that represents H.

We describe a 2-stage procedure to remove "useless" processors of H. A processor is removed if it is surrounded by high delays (stage 1) or few unremoved processors (stage 2). (When a processor is removed, its incident edges remain in the network.) For each depth k, we define two quantities D_k for "delay threshold" and m_k for "survival threshold". Note that D_k is larger than the average delay on a row/column in a depth-k array by a factor of $\Theta(\log n)$, and m_k is smaller than the number of processors in a depth-k array by a factor of $\Theta(\log n)$.

$$D_k = (c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right) \tag{4.1}$$

$$m_k = \left(\frac{1}{c \log n}\right)\left(\frac{n^2}{4^k}\right) \tag{4.2}$$

A constant c is specified later. We also define a maximum depth $k_{\max}$ such that when $k = k_{\max}$ the survival threshold m_k becomes 1.

$$k_{\max} = \log n - \frac{1}{2}\log c - \frac{1}{2}\log\log n. \tag{4.3}$$

- **Stage 1** From depth $k = k_{\max}$ down to depth 0, if the total delay on a row/column of a depth-k array exceeds the threshold D_k, then all the $\frac{n}{2^k}$ pro-

cessors on that row/column are removed.

- **Stage 2** From depth $k = k_{\max}$ down to depth 0, if the number of unremoved processors in a depth-k array is smaller than the threshold m_k, then all the processors in that array are removed. Moreover, we also remove processors so that the number of remaining processors in any depth-k array is an integer multiple of m_k.

Lemma 4.2.1 *At most $2n^2/c$ processors are removed in stage 1.*

Proof: The total delay of H is $2n^2 d_{\text{ave}}$. At most $\frac{2n2^k}{c \log n}$ depth-k rows and columns can have delay more than D_k. Since each depth-k row/column contains $\frac{n}{2^k}$ processors, at most $\frac{2n^2}{c \log n}$ processors are removed at depth k. There are $\log n$ depths, and so the lemma follows. □

Lemma 4.2.2 *At most n^2/c processors are removed at stage 2.*

Proof: Since there are 4^k depth-k arrays, at most $\frac{n^2}{c \log n}$ processors are removed at depth k. □

We label each array with the number of unremoved processors contained in it. By Lemmas 4.2.1 and 4.2.2, at most $3n^2/c$ processors of H are removed. Therefore, H is labeled with $c_1 n^2$, where $c_1 \geq 1 - (3/c)$. Any constant $c > 3$ works for our argument.

4.2.2 The Embedding

For clarity of presentation, we create an intermediate 2-dimensional array $\mathcal{G}$ that has size $\sqrt{c_1}n \times \sqrt{c_1}n$ and unit-delay edges only. We describe an algorithm EMBED that maps the processors of $\mathcal{G}$ one-to-one to the unremoved processors of H. The goal is to show that for any monotone path in $\mathcal{G}$ its image in H under EMBED has length $O(d_{\text{ave}} n \log^{5/2} n)$. As a result, H can simulate $\mathcal{G}$ with a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$. Obviously $\mathcal{G}$ can simulate G with constant slowdown.

EMBED partitions $\mathcal{G}$ into *regions* recursively, and each depth-k region of $\mathcal{G}$ corresponds to a depth-k array of H. The depth-0 region is the entire network $\mathcal{G}$. By

the construction of stage 2, $c_1 n^2$ (the number of processors in $\mathcal{G}$) is a multiple of m_0. Hence, $\mathcal{G}$ can be viewed as a collection of contiguous squares of size $\sqrt{m_0} \times \sqrt{m_0}$. We inductively assume that each depth-k region consists of contiguous squares of size $\sqrt{m_k} \times \sqrt{m_k}$, where m_k is defined in Equation (4.2). Each depth-k region R is partitioned into four depth $k+1$ regions R_1, R_2, R_3 and R_4 as follows. First, each $\sqrt{m_k} \times \sqrt{m_k}$ square of R is divided into four squares of size $\sqrt{m_{k+1}} \times \sqrt{m_{k+1}}$, where $\sqrt{m_{k+1}} = \sqrt{m_k}/2$. Suppose that R_i corresponds to a depth $k+1$ square of H that has z_i unremoved processors, then R_i has size z_i. By the construction of stage 2, z_i is a multiple of m_{k+1}. Hence, R_i can be formed as a collection of contiguous squares of size $\sqrt{m_{k+1}} \times \sqrt{m_{k+1}}$. Note that if z_i is 0, the the corresponding R_i is empty. (See Figure 4-3.)

At depth $k_{\max}$, each depth-$k_{\max}$ region consists of contiguous squares of size 1×1. EMBED maps the processors in a depth-$k_{\max}$ region of $\mathcal{G}$ to the unremoved processors in the corresponding depth-$k_{\max}$ array of H in an arbitrary one-to-one manner. Thus, we have a one-to-one mapping from the processors of $\mathcal{G}$ to the unremoved processors of H.

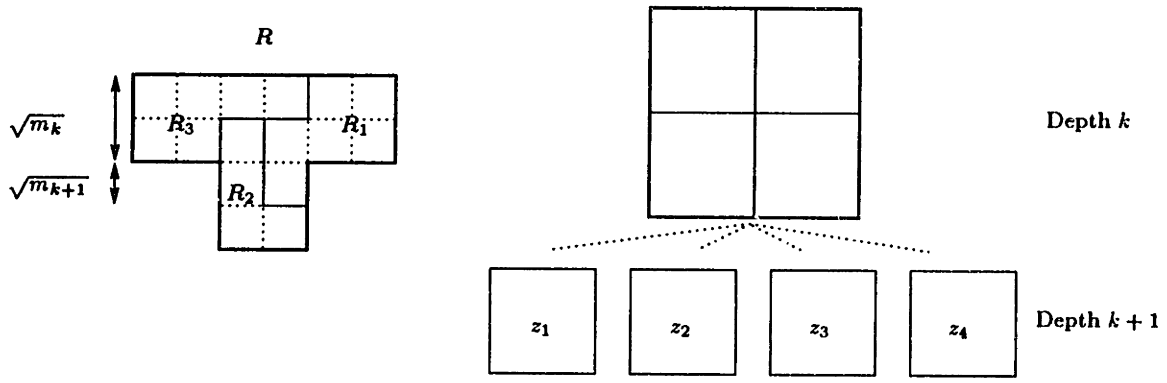


Figure 4-3: (Left) Depth-k region R and depth $k+1$ regions R_1, R_2 and R_3 of $\mathcal{G}$. (Right) Depth-k and $k+1$ arrays of H. Depth $k+1$ region R_i has size z_i, where z_i is the number of unremoved processors in the corresponding array of H. In this figure, $z_4 = 0$, and R_4 is therefore empty.

We also define the *depth-k boundaries* in $\mathcal{G}$ to be the borders of depth-k regions of $\mathcal{G}$. Note that the depth-k boundaries are at least $\sqrt{m_k}$ apart in both horizontal and vertical directions.

4.2.3 Bounding Monotone Path Length

In this section we bound the total delay on the image of P in H, where P is any monotone path in $\mathcal{G}$. Suppose a and b are two neighboring processors in $\mathcal{G}$, then their images a_H and b_H in H are connected by a *1-bend* route as follows. First, a_H is routed along its row to b_H's column and then routed to b_H along the column. We define a and b (resp. a_H and b_H) to be *k-related* if k is the largest integer such that a and b (resp. a_H and b_H) are in a same depth-k region (resp. depth-k array). We also define a_H and b_H to be *peers* of each other. Note that each unremoved host processor can have 4 peers.

Lemma 4.2.3 *Let P be any monotone path in $\mathcal{G}$. The image of P in H has a total delay of $O(d_{\text{ave}} n \log^{5/2} n)$ under* EMBED.

Proof: Let a and b be two neighboring processors on P and let a_H and b_H be their images. Suppose a and b (resp. a_H and b_H) are k-related. We first bound the length of the 1-bend route from a_H to b_H. By the construction of stage 1, the total delay on a depth-k row/column that contains a_H or b_H is at most D_k. Hence, the distance from a_H to b_H is at most $2D_k$.

We now bound the number of neighboring a's and b's that can be k-related. If $k < k_{\max}$, P must cross some depth $k+1$ boundary of $\mathcal{G}$ in traveling from a to b. Since P is monotone and the depth-k boundaries are $\sqrt{m_k}$ apart in both horizontal and vertical directions, P can cross the depth-k boundaries at most $\frac{2n}{\sqrt{m_k}}$ times. Hence, at most $\frac{2n}{\sqrt{m_{k+1}}}$ neighboring a's and b's on P can be k-related. This implies that the total delay incurred by k-related peers on the image of P is at most $2D_k \frac{2n}{\sqrt{m_{k+1}}}$, for $k < k_{\max}$. Obviously, at most $2n$ neighboring a's and b's can be $k_{\max}$-related. Summing over all depths, we conclude that the total delay on the image of P is at most $2D_{k_{\max}} \cdot 2n + \sum_{k<k_{\max}} 2D_k \frac{2n}{\sqrt{m_{k+1}}}$, which is $O(d_{\text{ave}} n \log^{5/2} n)$ by the definitions of D_k, m_k and $k_{\max}$. □

Hence, we can embed G on H such that $O(1)$ guest processors are mapped to each host processor and that the image in H of any monotone path in G has length $O(d_{\text{ave}} n \log^{5/2} n)$. Lemma 4.1.2 implies that H can simulate G with a slowdown

of $O(d_{\text{ave}} \log^{5/2} n)$. To improve the slowdown and achieve efficiency, we apply the idea of complementary slackness and use an $m \times m$ contiguous subarray of H for simulation as in FATRAY. Theorem 4.1.4 and Lemma 4.2.3 imply a slowdown of $O\left((d_{\text{ave}} m \log^{5/2} m)/n + n^2/m^2\right)$. By choosing m to be $\max\left\{n d_{\text{ave}}^{-1/3} \log^{-5/6} n, 1\right\}$, we have,

Theorem 4.2.4 *Network H with average delay d_{ave} can efficiently simulate G with a slowdown of $O\left(d_{\text{ave}}^{2/3} \log^{5/3} n\right)$.*

4.2.4 Bandwidth

The preceding analysis focuses entirely on the issue of latency and ignores bandwidth constraints. This does not present any problems if the link bandwidth available on the host array is $\Omega(\log^{3/2} n)$ times larger than that on the guest array. If the bandwidth of the host and guest arrays are comparable, however, and if the guest array is fully utilizing the bandwidth on its links then congestion becomes an issue. In this case, we may need to slow down the simulation by an additional factor of $O(\log^{3/2} n)$.

In Section 4.2.3, peers a_H and b_H are connected by a 1-bend route in H. To address the congestion issue, we present a more sophisticated method of connecting a_H and b_H such that each edge in H has $O(\log^{3/2} n)$ routes going through it and that the distance between a_H and b_H remains unchanged asymptotically.

We begin with some definitions. Recall that EMBED maps each depth-k region R_k of $\mathcal{G}$ to a depth-k array S_k of H. A depth-k row/column of S_k is *live* if it contains some unremoved host processors. A boundary point of S_k is live if it belongs to some live row or column of S_k. We first bound the number of connections from inside of S_k to outside of S_k in terms of the number of live rows and columns of S_k.

Lemma 4.2.5 *Consider any depth-k array, S_k, of H. The number of processors in S_k that have peers outside S_k is $O(x\sqrt{\log n})$, where x is the number of live rows and columns in S_k.*

Proof: Let z be the number of unremoved processors in S_k, then the number of live rows and columns is at least $\frac{z}{n/2^k}$. The number of host processors in S_k that

have peers outside S_k is proportional to the perimeter of R_k, the depth-k region that corresponds to S_k. By the construction of EMBED, R_k consists of squares of size $\sqrt{m_k} \times \sqrt{m_k}$. Hence, R_k has perimeter of $O(z/\sqrt{m_k})$, which is $O\left(\frac{z}{n/2^k}\sqrt{\log n}\right)$ by the definition of m_k in Equation (4.2). Our lemma follows. □

We now describe a recursive procedure that connects the peers. The following facts are used in our routing.

Fact 4.2.6 *Consider a routing problem on a square array of size $x \times x$.*

1. *If each node has $O(y)$ requests, then the routing can be done in 1 bend and $O(xy)$ congestion.*

2. *Let the nodes on the* cross *divide the square array into four $\frac{x}{2} \times \frac{x}{2}$ quadrants. If each boundary node and cross node have $O(y)$ requests and all other nodes have no requests, then the routing can be done in $O(1)$ bends and $O(y)$ congestion.*

Our recursive routing starts at depth $k = k_{\max}$. Consider all the depth-k arrays S_k. For all the peers that are k-related, we connect them through a 1-bend routing within S_k. Since S_k has size $\sqrt{\log n} \times \sqrt{\log n}$ and each host processor has at most 4 peers, the congestion caused by this 1-bend routing within S_k is $O(\sqrt{\log n})$ by item 1 of Fact 4.2.6. For all the processors that have peers outside S_k, we route them to live boundary points such that the following two conditions hold. First, each live boundary point of S_k receives $O(\sqrt{\log n})$ requests. This is possible because of Lemma 4.2.5. Second, the routing uses 1 bend and causes a congestion of $O(\log n)$ by item 1 of Fact 4.2.6.

We proceed recursively to depths $k < k_{\max}$. Consider all the depth-k arrays S_k. From the previous stage the host processors that are not connected to their peers are routed to some live boundary points of depth $k+1$ arrays. Hence, they are either on the boundary or on the cross of S_k, and $O(\sqrt{\log n})$ host processors are routed to the same location. For all the peers that are k-related, we connect them within S_k. Otherwise, we route them to the live boundary points of S_k such that each live point receives $O(\sqrt{\log n})$ requests (including those from all previous stages but have not yet

connected to their peers). This is possible by Lemma 4.2.5. In both cases, item 2 of Fact 4.2.6 implies that the routing can be done in $O(1)$ bends and that the congestion incurred is $O(\sqrt{\log n})$.

The congestion incurred at depth k, for $1 \leq k < k_{\max}$, is $O(\sqrt{\log n})$ and at depth $k_{\max}$ is $O(\log n)$. Since each of the depths uses the same underlying edges, the overall congestion is $O(\log^{3/2} n)$. The host processors are routed to live boundary points in $O(1)$ bends at each depth, and therefore the length incurred at depth k is $O\left(\frac{n}{2^k} d_{\text{ave}} \log n\right)$. Suppose that a_H and b_H are k-related, then the distance between them is $\sum_{k' \geq k} O\left(\frac{n}{2^{k'}} d_{\text{ave}} \log n\right)$, which remains $O\left(\frac{n}{2^k} d_{\text{ave}} \log n\right)$ as in Lemma 4.2.3. In summary,

Lemma 4.2.7 *In the above routing scheme the congestion is $O(\log^{3/2} n)$ on all edges of H. Furthermore, for any monotone path P in $\mathcal{G}$, the image of P in H has length $O(d_{\text{ave}} n \log^{5/2} n)$.*

4.3 Improved Bounds for Randomly-Arranged Delays

In this section, we show that the length of the longest monotone path in H is often short when the delays are randomly arranged. If M is the number of edges in an $n \times n$ array H, then for a given set of M delays with average d_{ave} the longest monotone path in H has length $O(n d_{\text{ave}})$ for most of the $M!$ permutations of the delays. That is, in the uniform distribution of the $M!$ permutations, the longest monotone path has length $O(n d_{\text{ave}})$ with high probability, and therefore the slowdown is $O(d_{\text{ave}})$ with high probability.

Without loss of generality we assume that d_{ave} is a constant. (For a nonconstant d_{ave}, each delay d is normalized to $\max\{d/d_{\text{ave}}, 1\}$. The normalized delays have average $O(1)$, and the total original delay on any monotone path is at most d_{ave} times the total normalized delay.) We divide the delays in H into $O(\log n)$ levels. Level ℓ contains the delays that are in the range of $[2^\ell, 2^{\ell+1})$, and level ℓ^+ contains the delays that are

at least 2^{ℓ}.

4.3.1 Shortcuts and Edge Coloring

If an edge with a large delay is surrounded by edges with small delays, we can route around this large delay. The intuition is that in a random permutation most long delays can be shortcut. For each edge f, we consider four edge-disjoint *alternate* paths that connect the two endpoints of f. (See Figure 4-4.) The 3×3 box that contains these four paths is called the *bounding box* of f. After the *shortcut*, the total delay on f equals the shortest alternate path length. For clarity, we shall refer to the delay before the shortcut as the *original delay* and the delay after the shortcut as the *shortcut delay.*

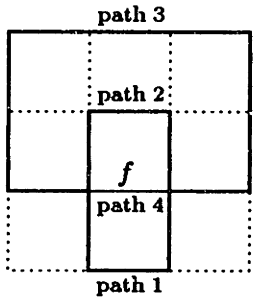


Figure 4-4: The bounding box and four edge-disjoint alternate paths for edge f.

For a given set of delays with a constant average, the number of level-ℓ^+ original delays is $O(n^2 2^{-\ell})$. Therefore, the probability for an edge f to have a level-ℓ^+ original delay is $O(2^{-\ell})$. However, shortcutting dramatically decreases this probability as the following lemma shows.

Lemma 4.3.1 *The probability for an edge f to have a level-ℓ^+ shortcut delay is $O(2^{-4\ell})$.*

Proof: If edge f has a shortcut delay from level ℓ^+, then the four alternate paths must each have an edge whose original delay is from level $(\ell-4)^+$. For a particular set of four edges to have level $(\ell-4)^+$ original delays, the probability is $\binom{B}{4}/\binom{M}{4}$, where B is the number of level $(\ell-4)^+$ original delays, and M is the number of edges

in H. Since there are $3 \cdot 3 \cdot 9 \cdot 1 = 81$ ways to choose four edges from four alternate paths, we derive the following from a union bound.

$$\Pr[\text{ Shortcut delay on } f \text{ is from level } \ell^+] \leq 81 \cdot \binom{B}{4} / \binom{M}{4}.$$

Our lemma follows from the observation that $B = O(n^2 2^{-\ell})$ since $d_{\text{ave}} = O(1)$, and that $M = \Theta(n^2)$. □

Unfortunately, these probabilities are *not* independent from edge to edge for two reasons. First, the arrangement of delays is a permutation of a given set of delays. This does not cause a problem however, as the analysis in Lemmas 4.3.2 and 4.3.2 will show. Intuitively, in a permutation if one edge has a large delay then other edges are less likely to have large delays. Second, the bounding boxes are not necessarily disjoint. To resolve this problem we introduce an *edge coloring*, so that any two distinct edges with the same color have edge-disjoint bounding boxes. Clearly, only a constant number of colors are needed.

We show in the following that, for *any* monotone path in H, the total delay incurred from the edges in one particular color group is $O(n)$ with high probability. Since there are $O(1)$ color groups, our results follows from a union bound. For each color group we consider two cases, edges with large shortcut delays and edges with small shortcut delays.

4.3.2 Large Delays

In this section we show that, with high probability, the total delay in H due to shortcut delays from large levels is $O(n)$. Therefore, any monotone path can only pick up $O(n)$ delay from these levels.

Lemma 4.3.2 *With probability* $1 - O(n^{-1})$, any *monotone paths pick up a total delay of* $O(n)$ *from levels* $\ell \geq L$, *where* $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.

Proof: By Lemma 4.3.1, the probability that one particular edge has a shortcut delay from level $(\frac{3}{4}\log n)^+$ is $O(n^{-3})$. Since H has $\Theta(n^2)$ edges, with probability

$1 - O(n^{-1})$ no edge in H has delay from level $(\frac{3}{4}\log n)^+$.

We show below that, with high probability, H has $O(\log^3 n)$ shortcut delays are from level L^+. Let $A = an^2$ be an upper bound the number of edges in one particular color group, where a is a constant. Since $d_{\text{ave}} = O(1)$, at most $B = bn^{3/2}\log^{1/2} n$ original delays can be from levels $(L-4)^+$, where b is a constant. We show that, with a small probability, more than $C = c\log^3 n$ edge delays are from level L^+, for a sufficiently large constant c.

For a particular set of C edges to have level-L^+ shortcut delays, at least 4 edges in each of these C bounding boxes have level $(L-4)^+$ original delays. For a particular set of four edges in each bounding box to have level $(L-4)^+$ original delays, the probability is at most $\binom{B}{4C}/\binom{M}{4C}$. This is true since all the C bounding boxes are edge-disjoint. There are at most $\binom{A}{C}$ ways to choose C edges whose shortcut delays are from L^+ and 81^C ways to choose four edges from each of the C boxes. We therefore derive the following from a union bound.

$$\begin{aligned} p &= \Pr[\text{ At least } C \text{ edges have level-}L^+ \text{ shortcut delays }] \\ &\le 81^C \binom{A}{C}\binom{B}{4C}/\binom{M}{4C}. \end{aligned} \tag{4.4}$$

We bound probability p with the inequalities,

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(\frac{ye}{x}\right)^x, \tag{4.5}$$

where $e = 2.718..$ is the base of the natural logarithm. By the definitions of A, B and C and the fact that $M \approx 2n^2$, we have,

$$p \le \left(\frac{81\cdot Ae}{C}\right)^C \left(\frac{Be}{M}\right)^{4C} = \left(\frac{81\cdot a\cdot b^4\cdot e^5}{2^4\cdot c\cdot \log n}\right)^{c\log^3 n}.$$

Let c be a sufficiently large constant, then the above probability is bounded by $O(n^{-1})$. Summing over all the $O(1)$ color groups, we conclude that with probability $1-O(n^{-1})$ H has no shortcut delays from level $(\frac{3}{4}\log n)^+$ and $O(\log^3 n)$ shortcut delays from level L^+. Hence, any monotone path picks up a total delay of $O(n^{3/4}\log^3 n) = O(n)$ from

levels $\ell \geq L$. □

4.3.3 Small Delays

In this section we show that the shortcut delay from small levels do not accumulate too much on any monotone path with high probability. In particular, with probability $1 - O(n^{-2})$, each monotone path picks up $O(n2^{-\ell})$ delay from each "small" level ℓ. Summing over all $O(\log n)$ "small" levels, we can conclude that each monotone path picks up a total of $O(n)$ small delays with probability $1 - O(n^{-1})$.

Consider a particular level $\ell < L$, where $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$. We divide H into m^2 squares of size $2^{2\ell} \times 2^{2\ell}$, where $m = n2^{-2\ell}$. There are $\binom{2m-1}{m}$ sequences of $2m-1$ squares that some monotone path could possibly go through. We call these sequences of $2m-1$ squares *monotone sequences.* If the total number of level-ℓ shortcut delays in each of these sequences is bounded, then the total level-ℓ shortcut delay that any monotone path picks up is also bounded.

Lemma 4.3.3 *With probability* $1 - O(n^{-2})$, any *monotone path picks up a total of* $O(n2^{-\ell})$ *delay from level-*ℓ *shortcut delays, where* $\ell < L$ *is one particular level and* $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.

Proof: Consider one particular monotone sequence of $2m-1$ squares of size $2^{2\ell} \times 2^{2\ell}$, where $m = n2^{-2\ell}$. Let random variable X be the number of level-ℓ shortcut delay in this sequence of squares, and let random variable X_i be the number of level-ℓ shortcut delays from the ith square in the sequence. We use a moment generating function argument to upper bound $X = X_1 + \ldots + X_{2m-1}$. We first bound the probability $\Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,]$. Let $A = a2^{4\ell}$ be an upper bound on the number of edges from one particular color group in each $2^{2\ell} \times 2^{2\ell}$ square, where a is a constant. Since $d_{\text{ave}} = O(1)$, at most $B = bn^2 2^{-\ell}$ original delays can be from level $(\ell - 4)^+$, where b is a constant. Let $k = \sum_{i=1}^{2m-1} k_i$. By applying the same logic as for Inequality (4.4), we have,

$$P = \Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,]$$

$$\leq 81^k \cdot \binom{A}{k_1} \cdot \ldots \cdot \binom{A}{k_{2m-1}} \cdot \binom{B}{4k} / \binom{M}{4k}.$$

By Inequality (4.5), the probability is bounded by,

$$\begin{aligned} P &\leq 81^k \cdot \left(\frac{Ae}{k_1}\right)^{k_1} \cdot \ldots \cdot \left(\frac{Ae}{k_{2m-1}}\right)^{k_{2m-1}} \cdot \left(\frac{Be}{M}\right)^{4k} \\ &= \prod_{i=1}^{2m-1} \left(\frac{81 \cdot a \cdot b^4 \cdot e^5}{2^4 \cdot k_i}\right)^{k_i}. \end{aligned} \tag{4.6}$$

We proceed to bound the expectation of e^X.

$$\begin{aligned} E[e^X] &= E[e^{X_1+\ldots+X_m}] \\ &= \sum_{k\geq 0} e^k \sum_{\sum k_i = k} \Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,] \\ &\leq \sum_{k\geq 0} \sum_{\sum k_i = k} \prod_{i=1}^{2m-1} \left(\frac{y}{k_i}\right)^{k_i}, \qquad \text{where } y = 81 \cdot a \cdot b^4 \cdot e^6 \cdot 2^{-4} \\ &\leq \sum_{k\geq 0} \sum_{\sum k_i = k} \prod_{i=1}^{2m-1} \frac{y^{k_i}}{k_i!} \\ &= e^{y(2m-1)}. \end{aligned}$$

The first inequality follows from Inequality (4.6), and the last equality follows from $e^{y(2m-1)} = \left(\sum_{j\geq 0} \frac{y^j}{j!}\right)^{2m-1}$. We use Markov's inequality to bound the probability that the total number of level-ℓ shortcut delays exceeds $\beta(2m-1)$ in this particular monotone sequence.

$$\Pr[\, X \geq \beta(2m-1)] = \Pr\left[\, e^X \geq e^{\beta(2m-1)}\right] \leq \frac{E[e^X]}{e^{\beta(2m-1)}} \leq e^{(y-\beta)(2m-1)}.$$

There are $\binom{2m-1}{m} < 2^{2m-1}$ monotone sequences. By a union bound, the probability that every sequence has fewer than $\beta(2m-1)$ level-ℓ shortcut delays is at least $1 - 2^{2m-1}e^{(y-\beta)(2m-1)}$. Let β be the constant $y+2$, then this probability is bounded by $1 - O(n^{-2})$, since $m = n/2^{2\ell} \geq \log n$. Therefore, every monotone path picks up a total of $O(n2^{-\ell})$ shortcut delays from level ℓ with probability $1 - O(n^{-2})$. □

Summing over all levels $\ell < L$ results in a total delay that is linear in n as desired.

Lemma 4.3.4 *With probability* $1 - O(n^{-1})$, *all the monotone paths pick up a total delay of* $O(n)$ *from levels* $\ell < L$ *for* $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.

For the case in which $d_{\text{ave}} = O(1)$, Lemmas 4.3.2 and 4.3.4 show that any monotone path has a total delay of $O(n)$ with high probability. For the case in which d_{ave} is nonconstant, the discussion at the beginning of section implies that,

Theorem 4.3.5 *Suppose* H *is a network with average delay* d_{ave}, *then with high probability every monotone path in* H *has delay* $O(nd_{\text{ave}})$.

To make the algorithm work-efficient, we use an $m \times m$ subarray of H of average delay at most d_{ave} to simulate G. Theorems 4.1.4 and 4.3.5 implies that the slowdown is $O(n^2/m^2 + d_{\text{ave}}m/n)$ with high probability. By choosing m to be ma $\{nd_{\text{ave}}^{-1/3}, 1\}$, we have,

Theorem 4.3.6 *Suppose the delays on network* H *are from a random permutation of a set of delays whose average is* d_{ave}, *then with high probability* H *can simulate* G *with slowdown* $O(d_{\text{ave}}^{2/3})$.

Congestion problems are not an issue here, since each edge of H is used $O(1)$ times by alternate paths in the shortcut process.

Chapter 5

Database Model

We switch our attention to the database model. As discussed in Chapter 2, simulation in the database model is more difficult than in the dataflow model. For algorithms such as STRIPE in Chapter 3 to work for the database model a host processor needs $\Theta(n)$ copies of the databases on average. This is unrealistic because of the memory requirement as well as the difficulty in updating the databases. We therefore develop new machinery for the database model. Contrary to the dataflow model, we make substantial use of redundant computation. Apart from the slowdown, another important parameter for the database model is *load*, which is the number of databases that a host processor copies.

The main contribution of this chapter is an algorithm called OVERLAP that simulates linear arrays in the database model with a small load and a small slowdown. Since OVERLAP is technically involved, we begin with a special case in Section 5.1 where the host linear array has delay d on all edges. The simulation in this special case is much simpler, and it conveys some intuition for using redundant computation in the general case. Section 5.2 presents OVERLAP in detail. The techniques are generalized to simulate linear and 2-dimensional arrays on general networks in Sections 5.3 and 5.4. Lastly in Section 5.5 we discuss the lower bounds on slowdown when each database is allowed a small number of copies.

5.1 A Special Case

In this section we consider a special case. Let G be a guest linear array with n processors and unit-delay edges, and let H be a host linear array with n processors and delay d on all edges. We use redundant computation, an approach that is not useful for the dataflow model, to achieve an optimal slowdown of $O(\sqrt{d})$.

Theorem 5.1.1 *In the database model, H can efficiently simulate G with a slowdown and a load of $O(\sqrt{d})$. This slowdown is optimal up to a constant factor.*

Proof: We consider two cases. If $n \leq \sqrt{d}$ then one host processor copies all the databases and carries out the entire computation by itself. Hence the load and the slowdown are n, which is $O(\sqrt{d})$. Otherwise, the first $\frac{n}{\sqrt{d}}$ host processors are used for the simulation. For $1 \leq j \leq \frac{n}{\sqrt{d}}$, processor p_j copies $3\sqrt{d}$ databases b_i and computes $3\sqrt{d}$ columns of pebbles (i,t), where $(j-2)\sqrt{d}+1 \leq i \leq (j+1)\sqrt{d}$ and $1 \leq t$. In this way each processor shares $\sqrt{d}$ databases with its right and left neighbors and each pebble is redundantly computed by three neighboring processors.

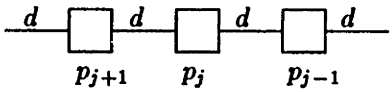
We show how to simulate the first $\sqrt{d}$ rows of pebbles created by G in $O(d)$ steps by H. Every subsequent $\sqrt{d}$ rows of pebbles are simulated in the same manner. The algorithm is demonstrated in Figure 5-1. For $1 \leq j \leq \frac{n}{\sqrt{d}}$ let,

$$
\begin{aligned}
P_j &= \{\text{Pebbles } (i,t): && 1 \leq t \leq \sqrt{d}, && -2\sqrt{d}+1 \leq i - j\sqrt{d} \leq \sqrt{d}\},\\
L &= \{\text{Pebbles } (i,t): && 1 \leq t \leq \sqrt{d}, && 1 \leq i - (j-2)\sqrt{d} \leq t\},\\
R &= \{\text{Pebbles } (i,t): && 1 \leq t \leq \sqrt{d}, && -t+1 \leq i - (j+1)\sqrt{d} \leq 0\},\\
A &= \{\text{Pebbles } ((j-2)\sqrt{d},t): && 1 \leq t \leq \sqrt{d}\},\\
B &= \{\text{Pebbles } ((j-1)\sqrt{d}+1,t): && 1 \leq t \leq \sqrt{d}\},\\
C &= \{\text{Pebbles } ((j\sqrt{d},t): && 1 \leq t \leq \sqrt{d}\},\\
D &= \{\text{Pebbles } ((j+1)\sqrt{d}+1,t): && 1 \leq t \leq \sqrt{d}\},\\
T &= P_j - (L \cup R).
\end{aligned}
$$

Processor p_j of H computes all the pebbles in P_j. First, p_j computes the pebbles in the trapezium T without communicating with its neighbors. There are $2d$ pebbles in

T and so this takes $2d$ steps. Next, p_j passes column B to processor p_{j-1} and receives column A from p_{j-1}. It also passes column C to processor p_{j+1} and receives column D from p_{j+1}. This communication takes $d + \sqrt{d} < 2d$ steps using pipelining. Processor p_j can now compute the pebbles in triangles L and R in d steps. It is important for p_j to compute the pebbles in L and R in order to continue the simulation of the next $\sqrt{d}$ rows of pebbles, since databases need to be updated. This presents a major difference between the dataflow and database models.

Hence, it takes at most $5d$ steps in total for processor p_j to compute every pebble in P_j. The next $\sqrt{d}$ steps of computation can be simulated in a similar fashion. The slowdown is therefore $O(\sqrt{d})$. Theorem 4.1.5 shows that $\Omega(\sqrt{d})$ is a lower bound on the slowdown.



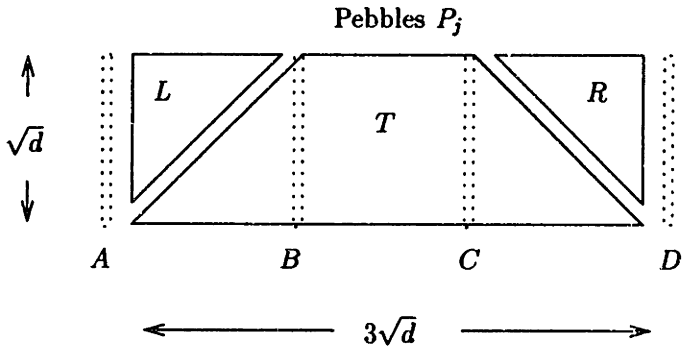


Figure 5-1: Simulating $\sqrt{d}$ steps of computation of G on H.

Note that during the computation of T, the pebbles in columns B and C can start to travel to the neighboring processors of p_j as soon as they are ready. Processor p_j can also start to compute triangles L and R before the entire columns of A and D are transfered. In this way, the communication time can be saved. Although it does not make a difference asymptotically in this case we take advantage of this observation in OVERLAP. □

5.2 Algorithm OVERLAP

To simulate a guest linear array on a host linear array with arbitrary delays we use an algorithm called OVERLAP. In OVERLAP, we remove host processors that are surrounded by high delays. The motivation of this step is similar to that of Section 4.2. For the remaining processors, we decide how much redundancy is needed for neighboring processors and how much computation each processor is able to carry out. During the simulation, some pebbles are redundantly computed to ensure that the communication is not too costly. We first obtain a slowdown of $O(d_{\text{ave}} \log^3 n)$, where d_{ave} is the average delay of H and n is the size of G and H, and later improve the slowdown to $O(\sqrt{d_{\text{ave}}} \log^3 n)$ while achieving work efficiency.

5.2.1 Removing Useless Processors

We recursively represent H using a binary tree, in which each node corresponds to a subarray of H. The root represents the entire array. The left and right children of the root represent the left and right halves of the array respectively. In general, a node at depth k of the binary tree corresponds to a subarray of H that contains $\frac{n}{2^k}$ processors. We refer to this subarray as a *depth-k interval.* The leaves represent the individual processors of H. (See Figure 5-2.)

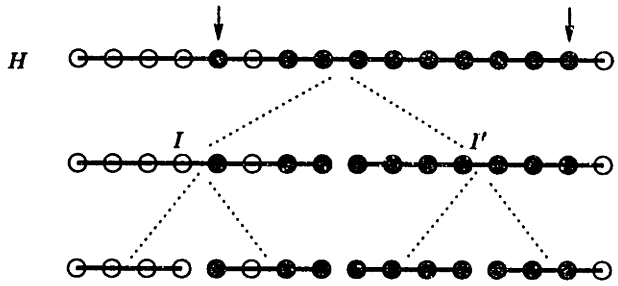


Figure 5-2: The binary tree that represents H. In this figure, unremoved processors of H are represented by black circles; removed processors are represented by white circles. Arrows indicate the endpoints of the root interval. Interval I has one live child and I' has two live children.

We describe a 2-stage process that removes the processors that are surrounded by high delays (stage 1) and the processors that are surrounded by few unremoved

processors (stage 2). During stage 2, we also label each *live* subarray, where a live subarray contains some unremoved processor. These labels indicate how many columns of pebbles the live subarrays are able to compute.

For every depth k, we define D_k to be the "delay threshold" and m_k to be the "overlap size" as follows. Note that D_k is larger than the average delay in a depth-k interval by a factor of $\Theta(\log n)$, and m_k is smaller than the number of processors in a depth-k interval by a factor of $\Theta(\log n)$. We shall use m_k to indicate the size of overlap between neighboring depth-k intervals, i.e. the number of columns of pebbles redundantly computed by both intervals.

$$D_k = (c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right) \tag{5.1}$$

$$m_k = \left(\frac{1}{c \log n}\right)\left(\frac{n}{2^k}\right). \tag{5.2}$$

As we shall see, any constant $c > 5/2$ works for our argument. We also define a maximum depth $k_{\max}$ such that when $k = k_{\max}$ the overlap size m_k becomes 1.

$$k_{\max} = \log n - \log \log n - \log c. \tag{5.3}$$

- **Stage 1** From depth $k = k_{\max}$ down to depth 0, if the total delay in a depth-k interval exceeds D_k, then all the processors in that interval are removed.

- **Stage 2** At depth $k = k_{\max}$, let I be a live depth-k interval and let x be the number of unremoved processors in I. If x is smaller than $2m_k$ then we remove all the remaining processors in I and I is no longer live. Otherwise, we label I with x.

  Suppose all the live depth-$(k+1)$ intervals are labeled. Now consider each live depth-k interval I. If I has two live children I_1 and I_2 that are labeled with x_1 and x_2, then let $x = x_1 + x_2 - m_{k+1}$. If I has one live child I_1 that is labeled with x_1, then let $x = x_1$. If $x < 2m_k$, we remove all the remaining processors in I and I is no longer live. Otherwise, we label I with x. We proceed to depth $k-1$ until reaching depth 0.

Lemma 5.2.1 *At most n/c processors are removed at stage 1.*

Proof: The total delay in the array H is nd_{ave}. At most $\frac{2^k}{c \log n}$ depth-k intervals can have delay more than D_k. Each depth-k interval contains $\frac{n}{2^k}$ processors and so at most $\frac{n}{c \log n}$ processors are removed at depth k. Since there are $\log n$ depths, at most n/c processors are removed at stage 1. □

Lemma 5.2.2 *The label on the root interval is at least $\left(1 - \frac{5}{2c}\right) n$ at stage 2.*

Proof: Before stage 2, the number of remaining processors in H is at least $(1 - 1/c)n$ by Lemma 5.2.1. At depth $k = k_{\max}$ of stage 2, the sum of the labels on the live depth-k intervals is at least $(1 - 1/c)n - 2m_k 2^k$, which is $(1 - 1/c)n - \frac{2n}{c \log n}$. At each depth $k < k_{\max}$, the sum of the labels on the live depth-k intervals decreases by at most $(2m_k + m_{k+1})2^k$, which is $\frac{5n}{2c \log n}$. Summing over all depths, we conclude that the label at the root interval is at least $\left(1 - \frac{5}{2c}\right) n$. □

5.2.2 Assigning Databases

For clarity of presentation, we first assume that G has n' processors, where n' is the label on the root interval of G and n' is a constant fraction of n by Lemma 5.2.2. We also assume the existence of pebbles $(0, t)$ and $(n' + 1, t)$, for all $t \geq 1$, which are known to H at time step 0. This ensures that each pebble computed by G is dependent on three pebbles.

Algorithm OVERLAP assigns one database to each remaining processor of H so that H has load one. In particular, a depth-k interval with label x is assigned x databases. The depth-0 interval, i.e. H, has all the databases $b_1, \ldots, b_{n'}$. We assume inductively that a depth-k interval I labeled x is assigned databases $b_{i+1}, \ldots, b_{i+x}$. If I has only one child I_1, then OVERLAP assigns $b_{i+1}, \ldots, b_{i+x}$ to I_1. If I has two children I_1 and I_2 that are labeled x_1 and x_2 respectively, then $x = x_1 + x_2 - m_{k+1}$ by the construction of stage 2. OVERLAP assigns $b_{i+1}, \ldots, b_{i+x_1}$ to interval I_1 and b_{i+x-x_2+1}, $\ldots$, b_{i+x} to I_2. Note that m_{k+1} databases, namely $b_{i+x-x_2+1}, \ldots, b_{i+x_1}$, are assigned to both I_1 and I_2. These m_{k+1} columns of pebbles will be redundantly computed

by both I_1 and I_2. At depth $k_{\max}$ each remaining processor of H is assigned one database.

5.2.3 The Simulation

In OVERLAP, H recursively simulates every $m_0 = \frac{n}{c \log n}$ rows of pebbles created by G as follows. If H, the depth-0 interval, has two live depth-1 intervals I_1 and I_2 as children, then I_1 and I_2 recursively compute the first $m_1 = m_0/2$ rows of pebbles and then repeat for the next m_1 rows. In particular, I_1 (resp. I_2) computes all the pebbles of the form (i,t), where I_1 (resp. I_2) owns database b_i and $1 \leq t \leq m_1$. Intervals I_1 and I_2 share m_1 databases and therefore redundantly compute these m_1 columns of pebbles. If H has one live child I_1, then I_1 recursively computes the first m_1 rows and then repeats for the second m_1 rows. At depth $k = k_{\max}$, each depth-k interval computes $m_k = 1$ row of pebbles. Theorem 5.2.3 explains the simulation in details.

Let us define a set of values $s_t^{(k)}$ for $0 \leq k \leq k_{\max}$ and $1 \leq t \leq m_k$, where the superscript k represents the depth of the recursion, and the subscript t represents the row number. Roughly speaking, $s_t^{(k)}$ corresponds to the time by which a depth-k interval computes its pebbles in the tth row. We are interested in the slowdown $s_{m_0}^{(0)}/m_0$, where $s_{m_0}^{(0)}$ corresponds to the time that H takes to simulate the first m_0 steps of computation by G. Recall that the delay threshold D_k defined in Equation (5.1) is an upper bound on the total delay in any live depth-k interval. The recurrences are as follows.

$$s_t^{(k)} = s_t^{(k+1)} + D_k \qquad \text{for} \qquad 1 \leq t \leq m_{k+1} \tag{5.4}$$

$$s_t^{(k)} = s_{t-m_{k+1}}^{(k)} + s_{m_{k+1}}^{(k)} \qquad \text{for} \quad m_{k+1}+1 \leq t \leq m_k \tag{5.5}$$

The base of the recurrence is defined to be,

$$s_{m_k}^{(k)} = s_1^{(k)} = 1 \qquad \text{for} \quad k = k_{\max}. \tag{5.6}$$

Let the *left endpoint* of interval I be the leftmost unremoved processor in I, and the

right endpoint be the rightmost unremoved processor in I. (See Figure 5-2.) For notational simplicity, we assume that I is the leftmost live depth-k interval and is assigned databases $b_1, \ldots, b_x$. Let $B_k = \{(i,t) : 1 \leq i \leq x, 1 \leq t \leq m_k\}$. The proof of the following theorem describes how algorithm OVERLAP performs the simulation.

Theorem 5.2.3 *For $1 \leq t \leq m_k$, if pebbles $(0,t)$ and $(x+1,t)$ are known by time step $s_t^{(k)}$ by the left and right endpoints of interval I respectively, then by time step $s_t^{(k)}$ every pebble (i,t) in B_k is computed by all the processors in interval I that have a copy of database b_i.*

Proof: We proceed by a backwards induction on k. At level $k = k_{\max}$, we have $m_k = 1$ and box B_k has size $x \times 1$. Since remaining processors of I have load one, each processor computes one pebble in B_k. By definition $s_1^{(k)} = 1$. Hence, the base of the induction holds.

Suppose that the inductive hypothesis is true for $k+1$. Note that the hypothesis can be applied to any depth $k+1$ interval. Let us concentrate on I, the leftmost live depth-k interval. Suppose I is labeled with x. There are two cases to consider.

Case 1: Suppose I has two live children I_1 and I_2 that are labeled with x_1 and x_2 respectively. By construction $x = x_1 + x_2 - m_{k+1}$. Let $B_{k+1} = \{(i,t) : 1 \leq i \leq x_1, 1 \leq t \leq m_{k+1}\}$. Let $y = x_1 - m_{k+1}$ and $B'_{k+1} = \{(i,t) : y+1 \leq i \leq x, 1 \leq t \leq m_{k+1}\}$. Let column C consist of pebbles (y,t) and column D consist of pebbles (x_1+1,t), where $1 \leq t \leq m_{k+1}$. Note that boxes B_{k+1} and B'_{k+1} have an overlap of width m_{k+1}, i.e. the m_{k+1} columns between C and D are common to both B_{k+1} and B'_{k+1}. (See Figure 5-3.) Two observations can be made from the inductive hypothesis.

- **Observation 1** For $1 \leq t \leq m_{k+1}$, every pebble (y,t) in column C can be computed by I_1 by time step $s_t^{(k+1)}$ *without any conditions on pebbles $(0,t)$ and (x_1+1,t).* Since C and D are m_{k+1} columns apart and $x_1 \geq 2m_{k+1}$ by the construction of stage 2, the pebbles in column C therefore do not depend on the pebbles $(0,t)$ and (x_1+1,t). (The dotted diagonal lines in Figure 5-3 show the dependencies of columns C and D.)

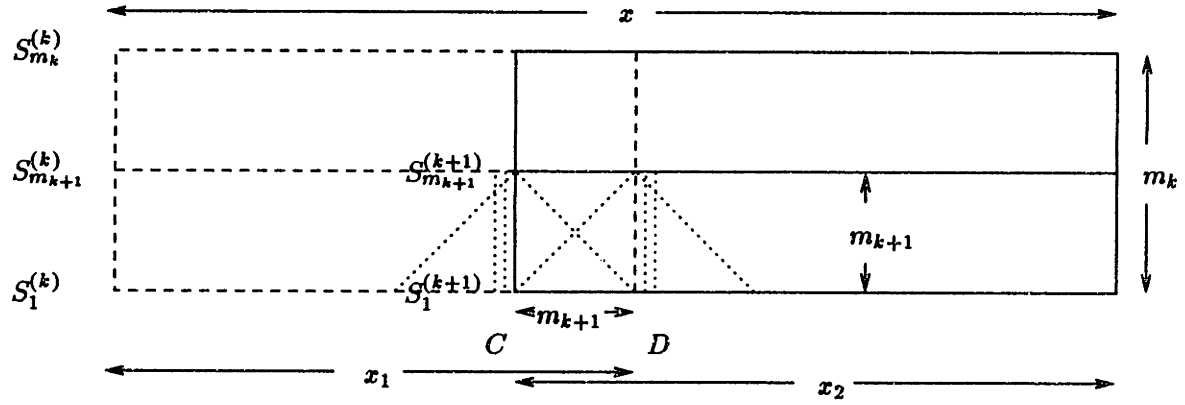


Figure 5-3: The box of pebbles B_{k+1} has size $x_1 \times m_{k+1}$ and is represented by the lower left box with a dashed boundary. B'_{k+1} has size $x_2 \times m_{k+1}$ and is represented by the lower right box with a solid boundary. B_k is the union of all four boxes. For interval I to compute every pebble in B_k, I_1 and I_2 (the live children of I) recursively compute B_{k+1} and B'_{k+1}. Once the bottom half of B_k is computed the top half is computed in a similar manner.

- **Observation 2** Let $z \geq 0$ be some constant. For $1 \leq t \leq m_{k+1}$, if the value of pebbles $(0,t)$ and (x_1+1,t) are known at time step $s_t^{(k+1)} + z$ by the left and right endpoints of interval I_1 respectively, then by time step $s_t^{(k+1)} + z$, every pebble (i,t) in B_{k+1} is computed. This is true because there is no difference between starting the simulation at time step z and at time step 0.

Similar statements can be made about the box B'_{k+1} and column D. Now suppose that the value of pebbles $(0,t)$ and $(x+1,t)$ are known at time step $s_t^{(k)}$ by the left and right endpoints of interval I respectively. Observation 1 and the inductive hypothesis imply that any pebble (y,t) in column C can be computed by I_1 by time $s_t^{(k+1)}$. Since the total delay in interval I is at most D_k then the left endpoint of interval I_2 can receive the pebble (y,t) (together with any relevant database changes) by time $s_t^{(k+1)} + D_k$ which equals $s_t^{(k)}$ (Equation (5.4)). Similarly, all of the pebbles in column D can be sent to the right endpoint of interval I_1 by time $s_t^{(k)}$. Since $s_t^{(k)}$ is greater than $s_t^{(k+1)}$ by a constant amount, namely D_k, for $1 \leq t \leq m_{k+1}$, Observation 2 and the inductive hypothesis imply that pebbles (i,t) in box B_{k+1} (resp. B'_{k+1}) are computed by I_1 (resp. I_2) by time $s_t^{(k)}$. Therefore, pebbles (i,t) in the bottom half of B_k are computed by time $s_t^{(k)}$.

Once the bottom half of B_k is simulated I simulates the top half in a similar

manner. Thus, pebbles (i,t) in the top half of B_k are computed by time $s^{(k)}_{m_{k+1}} + s^{(k)}_{t-m_{k+1}}$ which equals $s^{(k)}_t$ (Equation (5.5)).

Case 2: The case in which I has one live child is simpler. Let I_1 be the child of I. By construction, I_1 has label $x_1 = x$. By Observation 2 and the induction hypothesis, if the values of the pebbles $(0,t)$ and (x_1+1,t), for $1 \leq t \leq m_{k+1}$, are known at time steps $s^{(k)}_t$ by the left and right endpoints of interval I_1 respectively, then every pebble (i,t) in B_{k+1} (i.e. the bottom half of B_k) is computed by I_1 by time step $s^{(k)}_t$. Since intervals I and I_1 have the same remaining processors (and hence the same endpoints), the above statement holds for I. Interval I then computes the top half in the same manner. Thus, pebbles (i,t) in the top half of B_k are computed by time $s^{(k)}_{m_{k+1}} + s^{(k)}_{t-m_{k+1}}$ which equals $s^{(k)}_t$ (Equation (5.5)).

The inductive step is complete. Hence, given that the value of pebbles $(0,t)$ and $(x+1,t)$ are known at time step $s^{(k)}_t$ by the left and right endpoints of interval I all pebbles (i,t) in box B_k are computed by time step $s^{(k)}_t$. □

Recall that n' is the label of the tree root and n' is a constant fraction of n by Lemma 5.2.2. We have the following.

Theorem 5.2.4 *Suppose that guest linear array G has n' processors and the host linear array H has n processors and an average delay of d_{ave}. Algorithm* OVERLAP *simulates G with H such that the load on H is one and the slowdown is $O(d_{\text{ave}} \log^3 n)$.*

Proof: The load on H follows directly from the database assignment. The box B_0 contains all of the pebbles for the first m_0 steps of computations by G, where $m_0 = \frac{n}{c \log n}$. The root interval I_0 contains all the remaining processors of H. Since pebbles $(0,t)$ and $(n'+1,t)$ are available at time step 0 by assumption, Theorem 5.2.3 implies that I_0, i.e. H, computes the pebbles in box B_0 by time $s^{(0)}_{m_0}$. We derive $s^{(0)}_{m_0}$ from the recurrence of $s^{(k)}_t$ in Equations (5.4) and (5.5) and the definition of D_k in Equation (5.1).

$$s^{(0)}_{m_0} = 2^k s^{(k)}_{m_k} + 2kD_0 \qquad \text{for } k = k_{\max}. \tag{5.7}$$

Therefore, $s_{m_0}^{(0)} \leq \frac{n}{c \log n} + 2cd_{\text{ave}} n \log^2 n = O(d_{\text{ave}} n \log^2 n)$. Since $m_0 = \frac{n}{c \log n}$, the slowdown is $O(d_{\text{ave}} \log^3 n)$. □

5.2.4 Bandwidth

It is clear that the bandwidth required for the communication between depth-k intervals is at most the bandwidth of G. Therefore, congestion is not an issue if the bandwidth on H is at least $\log n$ times the bandwidth on G. If, however, the bandwidth on G and H are comparable then we need to pay an extra factor of $\log n$ in the slowdown.

5.2.5 Improvements

In this section we first modify OVERLAP to achieve work efficiency. So far each host processor is assigned at most 1 database, and the base of the recurrence is therefore $s_{m_k}^{(k)} = 1$ for $k = k_{\max}$ as defined in Equation (5.6). Observe that in Equation (5.7) the second term of $s_{m_0}^{(0)}$ dominates the first term. We can balance the two terms by increasing the value of $s_{m_k}^{(k)}$ for the base case, i.e. increasing the load on the host processors.

In particular, we use an m-processor subarray of the host linear array H to simulate an n-processor guest linear array, where $m = \max\left\{1, \frac{n}{d_{\text{ave}}} \log^{-3} n\right\}$ and the subarray has average delay at most d_{ave}. If $m = 1$, the slowdown and the load are both n. Otherwise, we carry out the 2-stage process to remove the useless processors of the m-processor subarray as as described in Section 5.2.1. The only difference is that the network size is m instead of n, and the variables such as D_k, m_k and $k_{\max}$ are also defined in terms of m. Each unremoved host processor is assigned $\Theta\left(\frac{n}{m}\right)$ databases, and hence the base case is $s_{m_k}^{(k)} = \Theta\left(\frac{n}{m}\right)$ for $k = k_{\max}$. We obtain,

$$s_{m_0}^{(0)} = \Theta\left(\frac{m}{c \log m} \cdot \frac{n}{m} + 2cd_{\text{ave}} m \log^2 m\right)$$

from Equation (5.7). Since $m = \frac{n}{d_{\text{ave}}} \log^{-3} n$, we have $s_{m_0}^{(0)} = O\left(n \log^{-1} \frac{n}{d_{\text{ave}}}\right)$. Since

$m_0 = \frac{m}{c \log m}$, the slowdown $s^{(0)}_{m_0}/m_0$ is $O(d_{\text{ave}} \log^3 n)$. This implies that the simulation is work preserving.

Theorem 5.2.5 *In the database model, an n-processor guest linear array can be efficiently simulated by an n-processor host linear array with a slowdown and a load of* $O\left(d_{\text{ave}} \log^3 n\right)$, *where the host has average delay* d_{ave}.

Combining Theorems 5.1.1 and 5.2.5 we can improve the slowdown to by a factor of $O(\sqrt{d_{\text{ave}}})$ while preserving efficiency. Suppose that G is an n-processor guest linear array, and H is an n-processor host linear array with average delay d_{ave}. We make use of an intermediate linear array H_0 that has a delay of d_{ave} on every edge. Theorem 5.1.1 implies that network H_0 can efficiently simulate G with a slowdown of $O(\sqrt{d_{\text{ave}}})$, where $\max\left\{\frac{n}{\sqrt{d_{\text{ave}}}}, 1\right\}$ processors of H_0 are used. In the simulation by H_0, every $O(d_{\text{ave}})$ steps of computation interleave with every $O(d_{\text{ave}})$ steps of communication. If we treat every $O(d_{\text{ave}})$ steps as one time unit, then H_0 acts like a guest linear array with unit-delay edges and H has a normalized average delay of $O(1)$. Theorem 5.2.5 implies that H can simulate H_0 with a slowdown of $O(\log^3 n)$. The combined slowdown is therefore $O(\sqrt{d_{\text{ave}}} \log^3 n)$. It is obvious that the combined load is $O(\sqrt{d_{\text{ave}}} \log^3 n)$. Theorem 5.2.5 is improved to the following.

Theorem 5.2.6 *In the database model, an n-processor guest linear array can be efficiently simulated by an n-processor host linear array with a slowdown and a load of* $O(\sqrt{d_{\text{ave}}} \log^3 n)$, *where the host has average delay* d_{ave}.

5.3 Simulating Linear Arrays on General Networks

We generalize algorithm OVERLAP to simulate a guest linear array on an arbitrary bounded-degree connected host network. Given a connected bounded-degree n-processor network H with average delay d_{ave}, we first find a linear array $\mathcal{H}$ that can be embedded one-to-one to H and has average delay d_{ave}. As discussed in Section 3.4 such $\mathcal{H}$ can be found, and $\mathcal{H}$ is used to carry out the simulation. Combined with Theorem 5.2.6, we obtain,

Theorem 5.3.1 *An n-processor guest linear array can be efficiently simulated by a connected bounded-degree n-processor host with a slowdown of* $O(\sqrt{d_{\text{ave}}}\log^3 n)$, *where the host has average delay* d_{ave}.

For the same reason as in Section 3.4 Theorem 5.3.1 does not hold when H has unbounded degree.

5.4 Simulating 2-Dimensional Arrays on General Networks

Our techniques can also be generalized to simulate a 2-dimensional array on any connected bounded-degree network.

Theorem 5.4.1 *In the database model, an* $n \times n$ *guest can be efficiently simulated by a bounded-degree host network with a slowdown of* $O(n\log^3 n + \sqrt{nd_{\text{ave}}}\log^3 n)$, *where the host has average delay* d_{ave}.

Proof: As discussed in section 3.4 there exists a linear array $\mathcal{H}$ such that $\mathcal{H}$ is embedded one-to-one in H and that $\mathcal{H}$ has average delay $O(d_{\text{ave}})$. The simulation of G on H will be performed by simulating G on $\mathcal{H}$. We first show how to simulate G on an intermediate linear array $\mathcal{H}_0$, where $\mathcal{H}_0$ has delay d_{ave} on all the edges. The size of $\mathcal{H}_0$ depends on the relative sizes of d_{ave} and n.

Case 1: If $d_{\text{ave}} < n$, then $\mathcal{H}_0$ has n processors, each of which simulates one column of processors of G. To simulate one step of G, a processor of $\mathcal{H}_0$ computes n pebbles and then communicates with both of its neighbors. The communication takes at most $n + d_{\text{ave}}$ steps, which is $O(n)$ steps. Hence the slowdown of $\mathcal{H}_0$ simulating G is $O(n)$. Also, in this simulation every $O(n)$ steps of computation interleave with every $O(n)$ steps of communication.

Since $d_{\text{ave}} < n$, if every $O(n)$ steps are treated as one time unit then $\mathcal{H}$ has a normalized average delay $O(1)$ and $\mathcal{H}_0$ acts like a guest linear array with unit-delay edges. Therefore, Theorem 5.2.6 implies that $\mathcal{H}$ can efficiently simulate $\mathcal{H}_0$ with a slowdown of $O(\log^3 n)$. The combined slowdown is therefore $O(n\log^3 n)$.

Case 2: If $d_{\text{ave}} \geq n$, then $\mathcal{H}_0$ has n/x processors, where $x = \sqrt{d_{\text{ave}}/n}$. Each processor of $\mathcal{H}_0$ simulates $3x$ columns of G, overlapping x columns with each neighbor. (The redundant computation used here is similar to that in Theorem 5.1.1.) To simulate x steps of G, each processor of $\mathcal{H}_0$ computes at most $3x^2n$ pebbles and then communicates with both of its neighbors. The communication takes at most $3x^2n + d_{\text{ave}}$ steps, which is $O(d_{\text{ave}})$ steps. Hence the slowdown of simulating every x steps is d_{ave}/x, which is $O(\sqrt{nd_{\text{ave}}})$. Also, in this simulation every $O(d_{\text{ave}})$ steps of computation interleave with every $O(d_{\text{ave}})$ steps of communication.

If every $O(d_{\text{ave}})$ steps is treated as one time unit, $\mathcal{H}$ has normalized average delay $O(1)$ and $\mathcal{H}_0$ acts like a linear array with unit-delay edges. If n/x processors of $\mathcal{H}$ are used to simulate $\mathcal{H}_0$, Theorem 5.2.6 implies a slowdown of $O(\log^3 \frac{n}{x})$, which is $O(\log^3 n)$. The combined slowdown is therefore $O(\sqrt{nd_{\text{ave}}} \log^3 n)$. □

The above technique can be applied to the dataflow model, where $\mathcal{H}_0$ simulates G in the same manner and $\mathcal{H}$ simulates $\mathcal{H}_0$ with a slowdown of $O(1)$ in both cases.

Theorem 5.4.2 *In the dataflow model, an $n \times n$ guest can be efficiently simulated by a bounded-degree host network with a slowdown of $O(n + \sqrt{nd_{\text{ave}}})$, where the host has average delay d_{ave}.*

5.5 Lower Bounds

In this section we discuss the impact on the slowdown of the simulation when the number of copies of each database is bounded and the load is a constant. We consider the case in which each database can have one copy and the case in which each database can have at most two copies. Notice that although we are restricting the number of copies of each database to either one or two, a particular processor in the host can have a copy of many databases.

For the case in which each database is allowed one copy we give an example to show that the slowdown can be $d_{\max}$. Let G and H_1 be n-processor guest and host linear arrays. Every $\sqrt{n}$-th edge of H_1 has a delay of $\sqrt{n}$ and all other edges have unit delay. Therefore, H_1 has an average delay of $O(1)$. If at most $\sqrt{n}$ processors of H_1

have copies of databases, then by a work argument the slowdown when H_1 simulates G is at least $\sqrt{n}$. Otherwise, there exist databases b_i and b_{i+1} such that they are assigned to processors p and q of H_1 respectively and that the delay between p and q is at least $\sqrt{n}$. Hence, for all time steps t, processor p cannot compute pebble (i,t) until $\sqrt{n}$ steps after q computes $(i+1,t-1)$, and q cannot compute $(i+1,t)$ until $\sqrt{n}$ steps after p computes $(i,t-1)$. This implies a slowdown of $d_{\max} = \sqrt{n}$, whereas d_{ave} is a constant. Note that the above argument makes no assumption on the load.

Theorem 5.5.1 *If each database can have at most one copy, the slowdown when simulating G by H_1 is $d_{\max}$.*

For the case in which each database is allowed at most two copies we construct a host network H_2 whose average delay is $O(1)$, but for which the simulation slowdown is $\Omega(\log n)$. Network H_2 is made up of $\Theta(n)$ processors and the edge delays are either 1 or d. The following is a recursive construction of H_2 in which we define a series of boxes. (See Figure 5-4.) We regard H_2 as a level-k box, where $k = \log \frac{n}{d}$. Network H_2

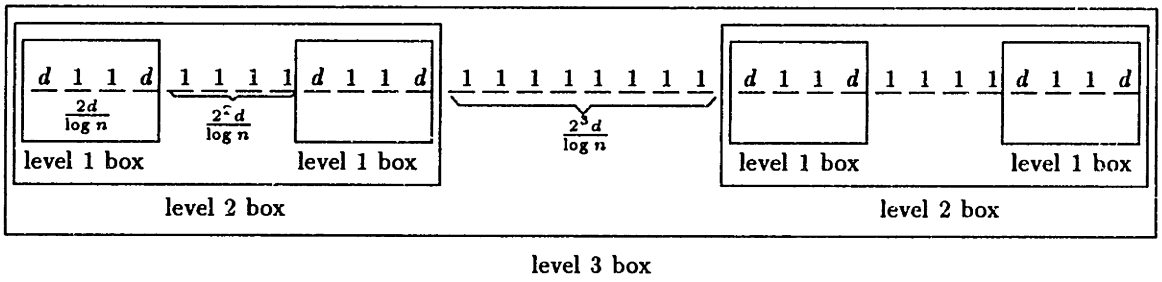


Figure 5-4: A level-3 box. Host network H_2 is a level-k box, where $k = \log \frac{n}{d}$.

consists of two level $k-1$ boxes that are connected by $\frac{2^k d}{\log n}$ edges of delay 1. In general, a level-ℓ box, for $1 \leq \ell \leq k$, consists of two level $\ell - 1$ boxes that are connected by $\frac{2^\ell d}{\log n}$ edges of delay 1. We say that these $\frac{2^\ell d}{\log n}$ processors are in a *segment*. A level-0 box consists of a single edge of delay d.

Let $d = \log n$. Since a level-ℓ box contains 2^ℓ edges of delay d and $\frac{2^\ell d \ell}{\log n}$ edges of delay 1, H_2 has $\Theta(n)$ processors and constant average delay d_{ave}. Furthermore,

Lemma 5.5.2 *If processors p and q are in two different segments I and J, then the delay between p and q is at least $\min\left\{\frac{u}{2}\log n, \frac{v}{2}\log n\right\}$, where u and v are the numbers*

of processors in segments I and J respectively. In particular, the delay between p and q is at least $d = \log n$.

Theorem 5.5.3 *If each database is allowed at most two copies and the load is a constant c, then the slowdown when simulating G by H_2 is $\Omega(\log n)$.*

Proof: We consider the following two cases.

Case 1: There exists some "overlap" in the database assignment. In particular, suppose databases b_i, $b_{i+1}, \ldots, b_{i+j}$ are assigned to processors in segment I and $b_{i+1}, \ldots, b_{i+j}$, b_{i+j+1} are assigned to segment $J \neq I$, for some $j \geq 1$. Suppose also that the other copy of b_{i+j+1} is assigned to $J' \neq I$ and the other copy of b_i is assigned to $I' \neq J$. Notice that pebbles of the form $(i + k, t)$, for $1 \leq k \leq j$, can only be computed by processors in segment I or J. Since the load is c, the number of processors in segment I is at least j/c. The same is true for segment J. We shall find a path of $4j$ pebbles such that either a delay of $O(j \log n)$ occurs, or a delay of $\log n$ occurs $O(j)$ times during the simulation. For simplicity we assume that j is even. The case in which j is odd is similar.

We use a triple (i, t, p_x) to say that processor p_x computes pebble (i, t), and we use expressions of the form $(i, t, p_x) \leftarrow (i - 1, t - 1, p_y)$ to indicate dependency. That is, processor p_x receives pebble $(i - 1, t - 1)$ from processor p_y before p_x computes (i, t). (Note that p_x may be the same as p_y.) Consider the computation of the following path of $4j$ pebbles, $\tau_1 \leftarrow \ldots \leftarrow \tau_{4j}$, where τ_k is a triple of the form,

$$\tau_k = \begin{cases} (i+k, t-k, p_k) & \text{for } k \in A, \text{ where } A = \{k : 1 \leq k \leq j\}, \\ (i+j+1, t-k, p_k) & \text{for } k \in B, \text{ where } B = \{k \text{ odd} : j < k \leq 2j\}, \\ (i+j, t-k, p_k) & \text{for } k \in C, \text{ where } C = \{k \text{ even} : j < k \leq 2j\}, \\ (i-k+3j, t-k, p_k) & \text{for } k \in D, \text{ where } D = \{k : 2j < k \leq 3j\}, \\ (i+1, t-k, p_k) & \text{for } k \in E, \text{ where } E = \{k \text{ even} : 3j < k \leq 4j\}, \\ (i, t-k, p_k) & \text{for } k \in F, \text{ where } F = \{k \text{ odd} : 3j < k \leq 4j\}. \end{cases}$$

This path goes backwards in time and zigzags during time steps k, for $k \in B \cup C \cup E \cup F$. (See Figure 5-5.)

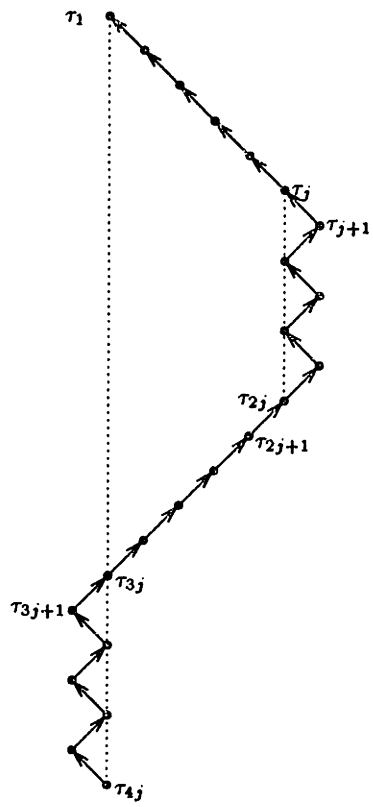


Figure 5-5: A path of $4j$ pebbles, where j is even.

By assumption processors p_k, for $k \in C \cup E$, can only belong to segment I or J. If processors p_k, for $k \in C \cup E$, do not belong to the same segment, then Lemma 5.5.2 implies a delay of $\frac{j}{2c} \log n$ for the communication between segments I and J. Hence, it takes more than $\frac{j}{2c} \log n$ steps to compute this path of $4j$ pebbles. Otherwise, processors p_k, for $k \in C \cup E$, all belong to segment I. Lemma 5.5.2 implies a delay of $\log n$ in computing every τ_k, for $j < k \leq 2j$. This is because processors p_k, for $k \in B$, cannot be in segment I by assumption. Similarly, if processors p_k, for $k \in C \cup E$, all belong to segment J, then there is a delay of $\log n$ in computing every τ_k, for $3j < k \leq 4j$. Hence, it takes more than $j \log n$ steps to compute this path of $4j$ pebbles.

We can repeat this argument for every $4j$ steps. Hence the slowdown is $\Omega(\log n)$.

Case 2: There exists no "overlapping" of the databases as in case 1. Let $b_i, \ldots, b_j$, for $j \geq i$, be the longest sequence of consecutive databases assigned to one segment. Call this segment I and the sequence of databases S_I. Notice that processors in I do

not have a copy of b_{i-1}. Let J be a segment that is assigned a copy of b_{i-1}. Let S_J be the sequence of consecutive databases such that b_{i-1} is a member of S_J and that each member of S_J has a copy in J. If b_i were a member of S_J, then either the database sequences S_J and S_I would produce the "overlapping" pattern sufficient for case 1 or S_J would be longer than S_I. This latter case contradicts with the definition of S_I. Hence, any segment that has a copy of b_{i-1} cannot have a copy of b_i. This implies that the processors computing the pebbles in the $(i-1)$st and ith column are at least $\log n$ delay apart by Lemma 5.5.2. Therefore, the slowdown is $\Omega(\log n)$. □

Part II

Dynamic Packet Routing

Chapter 6

Overview

6.1 Model and Problem

Consider a connection-oriented network $\mathcal{N}$ of arbitrary topology, on which a set of *sessions* are defined. Each session is specified by a source node, a destination node and a simple path connecting these two nodes. (A path is *simple* if it uses each edge at most once.) Packets are injected to the network $\mathcal{N}$ in sessions. A packet injected in session i arrives at the source node of session i, traverses its predetermined path, and then is absorbed at its destination node. The injection is constrained by a rate r_i, so that at most $tr_i + 1$ packets can be injected in session i during any t consecutive steps. Another parameter is the path length d_i, which is the number of edges on the path of session i.

We assume that all packets have the same size. We also assume that at each step at most one packet can traverse each edge. When two packets simultaneously contend for the same edge, one packet has to wait in a queue. During the routing, packets wait in two different kinds of queues. After a packet has been injected, but before it leaves its source, the packet is stored in an *initial queue*. Once the packet has left its source, during any time it is waiting to traverse an edge, the packet is stored in an *edge queue*. The *end-to-end delay* (delay for short) for a packet is the total time from the packet injection until it reaches its destination. This includes the total time the packet spends waiting in both types of queues, plus the time it takes to traverse the

edges.

Our goal is to minimize both the end-to-end delay for each packet and the size for all edge queues. In order to achieve delay guarantees and bounded queue sizes, it is necessary to require that for all edges e, the sum of the rates of the sessions that use edge e is at most 1. Throughout we assume that the sum of the rates of the sessions using any edge e is at most $1 - \epsilon$, for a constant $\epsilon \in (0, 1)$.

Our research focuses on the problem of timing the movements of the packets along their paths. A *schedule* specifies which packets move and which packets wait in queues at each time step. In particular, we concentrate on *template-based schedules*, where we define a fixed *template* for each edge in the network in advance. A template of size M is a wheel with M slots, each of which contains at most one token. Each token is affiliated with some session. All templates spin at the speed of one slot per time step. A session-i packet can traverse the edge only if a session-i token appears. (If more than one session-i packet is waiting, then the one that has been waiting the longest gets to move.) The template size and the associated tokens do not change over time. Therefore, even if the computation of the schedule is time-consuming, it only needs to be done once. Packets can then be scheduled indefinitely as long as the sessions do not change.

6.2 Lower Bounds

For all schedules, the path length d_i is an obvious lower bound, since it takes every session-i packet d_i steps to cross d_i edges.

It is easy to see that $\Omega(1/r_i)$ is an *existential* lower bound for all schedules. Consider n sessions with the same rate $r = (1 - \epsilon)/n$, each of which has the same initial edge. If a packet is injected in each session simultaneously, one of the packets requires $n = \Omega(1/r)$ steps to cross e. Hence, $\Omega(1/r_i + d_i)$ is an existential lower bound for all schedules.

For *any* given set of sessions, $\Omega(1/r_i)$ is a lower bound for some session i in template-based schedules. Suppose the total rates add up to $1 - \epsilon$ for some edge e,

then there always exist two session-i tokens separated by at least $(1-\varepsilon)/r_i$ slots on the template for edge e, for some i. Otherwise, there would be more tokens than slots on this template. If a session-i packet is injected just after the first token has passed, then this packet cannot cross edge e until $(1-\varepsilon)/r_i$ steps later. Hence, $\Omega(1/r_i)$ is a lower bound for all template-based schedules.

If the schedule is not restricted to being template-based, the scheduler is more powerful. The scheduler does not have to decide on a fixed schedule in advance, but rather can make a new decision at each step, based on the injections. In this case it is unknown if for *any* given set of sessions, $\Omega(1/r_i)$ is a lower bound.

6.3 Results

We present a simple distributed scheduler in Chapter 7. In a *distributed* scheme, each edge decides which packet to advance next without knowledge of the entire network. We obtain a delay bound of $O\left(\frac{1}{r_i} + d_i \log \frac{m}{r_{\min}}\right)$, where m is the number of edges in the network and $r_{\min}$ is the minimum injection rate over all sessions. While this bound is not optimal, it nevertheless conveys some intuition for our main result.

We also present an asymptotically-optimal template-based schedule, which allows every session-i packet to be delivered to its destination within $O(1/r_i + d_i)$ steps of its injection. This delay bound matches the lower bound of $\Omega(1/r_i + d_i)$ and therefore is optimal up to a constant factor. Our result improves upon previous work in several aspects.

- We prove that an additive delay bound of $O(1/r_i + d_i)$ is achievable. It is tempting to believe that a multiplicative bound of $O(d_i/r_i)$ is the best possible, since a session-i packet may need to wait $1/r_i$ steps in order to advance each edge. For example, Parekh and Gallager [47, page 148] give a bound of $2d_i/r_i$ under our model. Their scheme has the advantage of being simple and distributed.

- We provide a session-based delay guarantee without dropping any packets. This means that packets from sessions with short paths and high injection rates reach

their destinations fast. Some previous work states the delay bound in terms of $R = \max_i(1/r_i)$ and $D = \max_i d_i$ while allowing a packet loss probability of p. For example, Rabani and Tardos prove a bound of $O(R)+(\log^* p^{-1})^{O(\log^* p^{-1})}D+$ $poly(\log p^{-1})$ in [49]. This bound is improved to $O(R + D + \log^{1+\varepsilon} p^{-1})$ by Ostrovsky and Rabani [41]. Since their bounds are not session-based, a session with a long path or low injection rate affects the delay bounds for all sessions. The schemes of Rabani et al. are distributed, where knowledge of the entire network is not assumed, but each packet carries some information.

- An additional bonus of our result is constant-size edge queues. This is interesting because edge queues are much more costly than initial queues in practice. An initial queue is usually considered as a part of the sender's terminal, which has a relatively large amount of cheap and slow memory. An edge queue, however, is a part of a network element, which has expensive and fast memory.

- A consequence of our result is a packet-based bound for the static problem, which improves upon the $O(c + d)$ bound in [31]. Here, the congestion c is the maximum number of paths that cross an edge and the dilation d is the maximum path length. (In the static problem, all packets are present initially. See Chapter 8 for more details.) Suppose packet p follows a route of P_i, then p can be routed to its destination within $O(c_i+d_i)$ steps, where c_i is the maximum congestion along P_i and d_i is the number of edges on P_i. The result trivially follows from our result by creating a different session i for each packet p, and defining $r_i = (1 - \varepsilon)/c_i$.

In Chapter 8 we review the techniques in [31] and discuss how to adapt them to our routing problem. We prove the existence of a schedule with delay bound $O(1/r_i + d_i)$ in Chapter 9, and we present general approaches to constructing such a schedule in Chapter 10.

6.4 Generalization to the Leaky-Bucket Injection Model

Our results above can be generalized to bursty traffic streams that are *leaky-bucket regulated.* Here, each session i has a maximum burst size (or bucket size) of $b_i \geq 1$ and an average arrival rate of r_i. During any t consecutive time steps at most $r_i t + b_i$ session-i packets are injected. (Therefore, our injection model in Section 6.1 is the special case of the leaky-bucket model, in which the maximum burst size is 1.)

Leaky-bucket regulated injections allow traffic shaping. When session-i packets are injected, they first enter the session-i bucket at the source. These packets then leave the bucket one at a time at the rate of r_i. In this way, the end-to-end delay is separated into two components, delay in the bucket and delay in the network. Since delay in the bucket is upper bounded by b_i/r_i, the end-to-end delay is increased by at most b_i/r_i steps and the size of the edge queues is unchanged.

Chapter 7

Preliminaries

In this chapter we present some preliminary results. Section 7.1 proves a generic fact about "token sequences" for template-based schedules. Section 7.2 presents two lemmas for probabilistic analysis that will be used extensively throughout the thesis.

We also show a simple scheduler that, with high probability, generates a schedule that achieves a delay bound of $O\left(\frac{1}{r_i} + d_i \log \frac{m}{r_{\min}}\right)$, where m is the number of edges and $r_{\min} = \min_i r_i$. We begin with a centralized scheme in Section 7.3 and convert it to a distributed scheme in Section 7.4. This preliminary result is substantially simpler to prove than the optimal result of $O(1/r_i + d_i)$ because of the relaxed bounds. Nevertheless, it illustrates the basic ideas necessary to prove the main result.

7.1 Token Sequences

Throughout the thesis we define template-based schedules in terms of *token sequences*. A token sequence for session i consists of d_i session-i tokens, $\mathcal{K}_1, \ldots, \mathcal{K}_{d_i}$, one from each template along the session-i path. If $\mathcal{K}_{j+1}$ appears x_j steps after $\mathcal{K}_j$, then x_j is the *token lag* for these two tokens and $\sum_{j=1}^{d_i-1} x_j$ is the end-to-end delay for this token sequence. The token sequences for each session i form a partition of all the session-i tokens.

In the following we show that, in any template-based schedule, bounding the delay for token sequences is sufficient to bound the packet delays and that bounding the

token lag is sufficient to bound the edge queues. Our proof relies on Lemma 7.1.1. A vector $\vec{v} = [v_1, v_2, \ldots, v_n]$ is *sorted* if $v_1 \leq v_2 \leq \ldots \leq v_n$. We define $perm(\vec{v})$ to be a sorted vector whose components form a permutation of the components of $\vec{v}$. We also use the notation $\vec{u} < \vec{v}$ to indicate that the jth component of $\vec{u}$ is smaller than the jth component of $\vec{v}$ for each j.

Lemma 7.1.1 *Let $\vec{u} = [u_1, u_2, \ldots, u_n]$ and $\vec{v} = [v_1, v_2, \ldots, v_n]$ be two vectors, each of which consists of n distinct numbers. If $\vec{u} < \vec{v}$, then,*

1. *$perm(\vec{u}) < perm(\vec{v})$;*
2. *If $\vec{v} < \vec{u} + \vec{z}$, then $perm(\vec{v}) < perm(\vec{u}) + \vec{z}$, where $\vec{z} = [z, \ldots, z]$ is a vector of n z's for a scalar z.*
3. *Let $|\vec{v}|$ represent the maximum component of $\vec{v}$, then $|perm(\vec{v}) - perm(\vec{u})| \leq |\vec{v} - \vec{u}|$.*

Proof: Without loss of generality, we assume $\vec{u}$ is sorted, i.e. $\vec{u} = perm(\vec{u})$. We also assume $perm(\vec{v}) = [v_{\sigma(1)}, \ldots, v_{\sigma(n)}]$, where σ represents the sorted permutation of $\vec{v}$.

1. Let us compare u_j and $v_{\sigma(j)}$. There are two cases to consider. If $j \leq \sigma(j)$, then $u_j \leq u_{\sigma(j)} < v_{\sigma(j)}$. These inequalities hold since $\vec{u}$ is sorted by assumption and $\vec{u} < \vec{v}$. If $j > \sigma(j)$, then there exists $j' \geq j$ such that $v_{j'} \leq v_{\sigma(j)}$. (Otherwise, for all $j' \geq j$, $v_{j'} > v_{\sigma(j)}$. However, only $n - j$ components of $\vec{v}$ can be greater than $v_{\sigma(j)}$.) Combining the fact that $\vec{u}$ is sorted and $\vec{u} < \vec{v}$, we have $u_j \leq u_{j'} < v_{j'} \leq v_{\sigma(j)}$. Therefore, $perm(\vec{u}) < perm(\vec{v})$ in both cases.
2. Since $perm(\vec{u}+\vec{z}) = perm(\vec{u})+\vec{z}$ for $\vec{z} = [z, \ldots, z]$, Property 1 implies $perm(\vec{v}) < perm(\vec{u} + \vec{z}) = perm(\vec{u}) + \vec{z}$.
3. Suppose $|perm(\vec{v}) - perm(\vec{u})| = v_{\sigma(j)} - u_j$. There are two cases to consider. If $v_{\sigma(j)} \leq v_j$, then $v_{\sigma(j)} - u_j \leq v_j - u_j$, which implies $|perm(\vec{v}) - perm(\vec{u})| \leq |\vec{v} - \vec{u}|$. If $v_{\sigma(j)} > v_j$, then there exists $j' < j$ such that $v_{\sigma(j)} \leq v_{j'}$. (Otherwise, for all

$j' \leq j$, $v_{\sigma(j)} > v_{j'}$. However, only $j-1$ components of $\vec{v}$ can be smaller than $v_{\sigma(j)}$.) Since $u_{j'} < u_j$ by the assumption that $\vec{u}$ is sorted, we have $v_{\sigma(j)} - u_j \leq v_{j'} - u_{j'}$, which implies $|perm(\vec{v}) - perm(\vec{u})| \leq |\vec{v} - \vec{u}|$. Property 3 follows.

□

We are ready to transform a token-sequence-based bound into a packet-based bound. Although it might seem straight-forward, the difficulty is that a packet is unable to identify a token sequence. This means if a session-i token appears, then the session-i packet that has been waiting the longest has to move. The first token in a token sequence is called an *initial token.*

Theorem 7.1.2 *Consider any template-based schedule. If the end-to-end delay for each session-i token sequence is bounded by X, then each session-i packet reaches its destination within X steps after it obtains an initial token. If the token lag is bounded by x for* all *token sequences for* all *sessions, then the edge queue size is also bounded x.*

Proof: It suffices to show the following. For any $y \geq 1$, consider the first y session-i packets injected. After obtaining its initial token, each of these y packets reaches the destination within X steps and it waits at most x steps to advance each edge.

Let T_{k1} be the time that the kth packet catches an initial token $\mathcal{K}_k$ and advances its first edge. Let T_{kj} be the time that the kth packet would cross the jth edge if it followed the token sequence initiated at $\mathcal{K}_k$. Note that T_{kj} is not necessarily the time that the kth packet crosses the jth edge in a template-based schedule. However, T_{kj} does represent a time that a token would appear on the jth edge. We have $T_{11} < T_{21} < \ldots < T_{y1}$, and $T_{k1} < T_{k2} < \ldots < T_{kd_1}$ for $1 \leq k \leq y$.

We first apply Property 1 of Lemma 7.1.1 to show that packets 1 through y are able to cross the jth edge by times $perm(T_{1j}, T_{2j}, \ldots, T_{yj})$, for $1 \leq j \leq d_i$. Take an example of the second edge. Let $perm(T_{12}, T_{22}, \ldots, T_{y2}) = [T_{\sigma(1),2}, T_{\sigma(2),2}, \ldots, T_{\sigma(y),2}]$, where σ represents the sorted permutation. Property 1 of Lemma 7.1.1 implies $[T_{11}, T_{21}, \ldots, T_{y1}] < [T_{\sigma(1),2}, T_{\sigma(2),2}, \ldots, T_{\sigma(y),2}]$. Since packet 1 has left its first edge by

time T_{11} and an unused token for the second edge appears by $T_{\sigma(1),2}$, packet 1 is able to advance its second edge by $T_{\sigma(1),2}$. Since packet 1 has left by $T_{\sigma(1),2}$, packet 2 is able to obtain an unused token by $T_{\sigma(2),2}$ and advance its second edge. Similar reasoning applies to packets 3 through y for the second edge. Inductively, packets 1 through y are able to advance their last edge by $perm(T_{1d_i}, T_{2d_i}, \ldots, T_{yd_i})$. This quantity is bounded by $[T_{11} + X, T_{21} + X, \ldots, T_{y1} + X]$ by Property 2 of Lemma 7.1.1. Hence, all the session-i packets reach their destination within X steps after they obtain the initial tokens.

Let us bound the queue size now. Consider the jth edge, where $1 \leq j \leq d_i$. Suppose packet k, for $1 \leq k \leq y$, uses token $\mathcal{K}_{ij}$ to cross its $j$the edge at time t_{kj}. Let $\mathcal{K}_{k,j+1}$ be the $(j+1)$st token on the same token sequence as $\mathcal{K}_{kj}$, and let $t_{k,j+1}$ be the time that $\mathcal{K}_{k,j+1}$ appears. (Note that $\mathcal{K}_{kj}$ is not necessarily on the same token sequence as the initial token that packet k used to cross its first edge, and that $\mathcal{K}_{k,j+1}$ is not necessarily the token that packet k would use to cross the $(j+1)$st edge.) Since $t_{kj} < t_{k,j+1}$, Property 1 of Lemma 7.1.1 and our argument for the delay bound above imply that packets 1 through y are able to cross the $(j+1)$st edge by $perm(t_{1,j+1}, t_{2,j+1}, \ldots, t_{y,j+1})$. Property 3 of Lemma 7.1.1 shows that $|perm(t_{1,j+1}, t_{2,j+1}, \ldots, t_{y,j+1}) - [t_{1j}, t_{2j}, \ldots, t_{yj}]||$ is bounded by x, the token lag. Hence, a packet waits at most x steps to advance each edge once it obtains an initial token. □

7.2 Lemmas for Probabilistic Analysis

Throughout the construction of our schedules, we use the Lovász Local Lemma [54, pages 57-58] and a Chernoff Bound [10] for probabilistic analysis. We include them here for easy reference.

[Lovász Local Lemma] *Let $E_1, \ldots, E_n$ be a set of "bad events" each occurring with probability at most p and with dependence at most d (i.e. every bad event is*

mutually independent of some set of $n - d$ other bad events). If $4pd < 1$, then

$$\Pr\left[\bigcap_{i=1}^{n} \bar{E}_i \right] > 0.$$

In other words, no bad event occurs with a nonzero probability.

[Chernoff Bound] *Let X_i be n independent Bernoulli random variables with probability of success p_i. Let $X = \sum_{i=1}^{n} X_i$ and let the expectation $\mu = \sum_{i=1}^{n} p_i$. Then for $0 < \delta < 1$, we have,*

$$\Pr[\ X > (1+\delta)\mu\ ] \leq e^{-\delta^2\mu/3}.$$

We also prove a variation of the Chernoff bound.

Lemma 7.2.1 *Let X_i be n independent Bernoulli random variables with probability of success p_i. Let $X = \sum_i^n X_i$ and the expectation $E[X] = \sum_i^n p_i$. Then for $u \geq E[X]$ and $0 < \delta < 1$, we have,*

$$\Pr[\ X > (1+\delta)u\ ] \leq e^{-\delta^2 u/3}.$$

Proof: We prove the lemma by amplifying the success probabilities. If $u \geq n$, then $\Pr[\ X \geq (1+\delta)u\ ] = 0$ and we are done. Otherwise, let p_i' be a value such that $p_i \leq p_i' \leq 1$ and $\sum_i^n p_i' = u$. We have,

$$\begin{aligned} & \Pr[\ X > (1+\delta)u \mid \text{success probabilities } p_1, \ldots, p_n\ ] \\ \leq\ & \Pr[\ X > (1+\delta)u \mid \text{success probabilities } p_1', \ldots, p_n'\ ]. \end{aligned}$$

The Chernoff bound implies that the above probability is bounded by $e^{-\delta^2 u/3}$. □

7.3 A Simple Centralized Scheduler

We now present a centralized scheduler that achieves a delay bound of $O\left(\frac{1}{r_i} + d_i \log \frac{m}{r_{\min}}\right)$ and edge queues of size $O\left(\log \frac{m}{r_{\min}}\right)$.

7.3.1 Template Size

We first decide the template size M. Roughly speaking, M needs to be sufficiently large so that enough tokens can be placed to accommodate arrivals from all sessions every M steps. We express the injection rate for session i in terms of s_i/ℓ_i, a fraction lightly larger than r_i. If M is the least common multiple of ℓ_i for all i, then we can place s_i session-i tokens every ℓ_i consecutive slots on each template along the path of session i. The quantities of ℓ_i and s_i are defined as follows.

$$\ell_i = 2^{\lceil \log \frac{2}{\varepsilon r_i} \rceil} \tag{7.1}$$

$$s_i = \lfloor \ell_i r_i (1 + \varepsilon/2) \rfloor. \tag{7.2}$$

In other words, ℓ_i is the smallest power of 2 that is larger than or equal to $2/(\varepsilon r_i)$, and s_i is the largest integer that is less than or equal to $\ell_i r_i (1 + \varepsilon/2)$.

Lemma 7.3.1 *Let $\hat{r}_i = s_i/\ell_i$. We have the following properties for $\hat{r}_i$.*

1. *$r_i \leq \hat{r}_i$ for each session i;*
2. *$\sum_{i \in S_e} \hat{r}_i \leq 1 - \varepsilon/2$ for each edge e, where S_e is the set of sessions that cross edge e.*

Proof: We first show,

$$\ell_i r_i \leq s_i \leq \ell_i r_i (1 + \varepsilon/2). \tag{7.3}$$

The difference between the lower bound and the upper bound is $\ell_i r_i \varepsilon/2$, which is at least 1 by the definition of ℓ_i. Therefore, there is an integer in the range of $[\ell_i r_i, \ell_i r_i (1 + \varepsilon/2)]$, and s_i is such an integer by definition.

Inequality 7.3 implies $r_i \leq \hat{r}_i \leq r_i(1+\varepsilon/2)$. Property 1 is immediate. Given $\sum_{i \in S_e} r_i \leq 1-\varepsilon$, we have $\sum_{i \in S_e} \hat{r}_i \leq (1-\varepsilon)(1+\varepsilon/2) < 1-\varepsilon/2$. Property 2 follows. □

We now define the template size M to be $\max_i \ell_i$, which is $\Theta\left(\frac{1}{r_{\min}}\right)$. Since all the ℓ_i's are powers of 2, M is also the least common multiple of the ℓ_i's.

7.3.2 Token Placement

We describe a procedure to place the tokens for all sessions. For each session i, we first place s_i *initial tokens* in one slot every ℓ_i slots on the template that corresponds to the first edge of session i. We then delay each initial token of session-i by an amount chosen uniformly and independently at random from $[L+1, L+\ell_i]$, where

$$L = 2^{\lceil \log(\frac{\alpha}{2}\log(mM)) \rceil}, \tag{7.4}$$

for a constant α. In other words, L is a power of 2 that is greater than or equal to $\frac{\alpha}{2}\log(mM)$. As we shall see, this is enough randomness to spread out the tokens. For every session-i token a placed on the template corresponding to the jth edge, we place a session-i token b on the template corresponding to the $(j+1)$st edge such that b appears exactly $2L$ steps after a. In this way, we have partitioned all the session-i tokens into $M\hat{r}_i$ sequences, where each token sequence has d_i tokens and two neighboring tokens in each sequence are $2L$ apart. In the following we show that the tokens cannot be too clustered.

Lemma 7.3.2 *At most L tokens appear in any consecutive L slots on any template with probability at least $1 - 1/(mM)$, where L is defined in (7.4) for a sufficiently large constant α.*

Proof: Since s_i initial tokens for session-i are placed in one slot every ℓ_i slots and each is delayed by an amount chosen independently and uniformly at random from $[L+1, L+\ell_i]$, the expected number of session-i tokens in a single slot is s_i/ℓ_i, which is $\hat{r}_i$. Hence by linearity of expectations and Property 2 of Lemma 7.3.1, the expected

number of tokens over all sessions in L consecutive slots is $\sum_i \hat{r}_i L \leq (1-\varepsilon/2)L$. For a particular interval of L consecutive slots on a particular template, let the random variable X be the number of tokens in these slots. Whether or not a token lands in these L slots is a Bernoulli event. Since the delays to the initial tokens are chosen independently and all session paths are simple, these Bernoulli events are independent. Since $E[X] \leq (1-\varepsilon/2)L$, we have the following by Lemma 7.2.1.

$$\Pr[\ X > L\ ] \leq \Pr[\ X > (1+\varepsilon/2)(1-\varepsilon/2)L\ ] \leq e^{-\varepsilon^2(1-\varepsilon/2)L/12}.$$

In m templates there are at most mM intervals of L consecutive slots. Therefore, by a union bound the probability that more than L tokens appear in *any* L consecutive slots is bounded by,

$$mM \Pr[\ X > L\ ] \leq mMe^{-\varepsilon^2(1-\varepsilon/2)L/12} = mMe^{-\varepsilon^2(1-\varepsilon/2)\alpha \log(mM)/24}.$$

By choosing a sufficiently large constant α, we can bound the above probability by $1/(mM)$. □

If the first pass of the delay insertion does not produce a token assignment that satisfies the condition of at most L tokens every L slots, we simply try another pass until the condition is met.

7.3.3 Smoothing

In order to guarantee one token per slot we carry out a *smoothing process.* Since there are at most L tokens in any consecutive L slots, we partition each template into intervals of L consecutive slots and arbitrarily place at most one token in each slot within each interval. (Note the template size M is multiple of L, since M and L are both powers of 2.) Recall we have defined token sequence for each session in the token placement process.

Lemma 7.3.3 *Let $\mathcal{K}_1, \ldots, \mathcal{K}_{d_i}$ be any token sequence for session i, then after the smoothing process we have,*

1. *Token $\mathcal{K}_j$ appears after $\mathcal{K}_{j-1}$, for $1 < j \leq d_i$;*

2. *The end-to-end delay of the token sequence is bounded by $2d_iL + 2L$ and the token lag is bounded by $4L$.*

Proof: Before the smoothing, $\mathcal{K}_j$ appears exactly $2L$ steps after $\mathcal{K}_{j-1}$ for $1 < j \leq d_i$, i.e. the token lag is $2L$. Since the smoothing process shifts each token by at most $L-1$ slots, $\mathcal{K}_j$ still appears after $\mathcal{K}_{j-1}$ after the smoothing. The token lag therefore increases to at most $4L$. The end-to-end delay for the token sequence increases from $2d_iL$ to at most $2d_iL + 2L$ due to the shift of the first and the last tokens. □

Theorem 7.3.4 *With high probability, the above randomized centralized scheme generates a template-based schedule that produces a delay bound of $O\left(\frac{1}{r_i} + d_i \log \frac{m}{r_{\min}}\right)$ and edge queues of size $O\left(\log \frac{m}{r_{\min}}\right)$.*

Proof: We first show that each session-i packet, p, is able to catch an initial token within $2L + 2\ell_i$ steps of its injection. Before the initial session-i tokens are delayed, we have exactly s_i tokens every ℓ_i slots. Since at most s_i session-i packets can be injected during ℓ_i steps, packet p would be able to obtain an initial token, say $\mathcal{K}$, in fewer than ℓ_i steps if the tokens were not delayed or shifted. Let p be injected at time t and let $\mathcal{K}$ appear at T before $\mathcal{K}$ is delayed and shifted, then $t \leq T < t + \ell_i$. Each initial token is delayed by an amount in the range of $[L+1, L+\ell_i]$ during the token placement process and shifted by at most $L-1$ slots during the smoothing process. Therefore, after the smoothing process $\mathcal{K}$ appears after t but before $t + 2L + 2\ell_i$.

By Theorem 7.1.2 and Lemma 7.3.3, any session-i packet p is able to reach its destination within $2d_iL+2L$ steps after it obtains its initial token. Therefore, the end-to-end delay for session-i packets is $(2L+2\ell_i)+(2d_iL+2L)$, which is $O\left(\frac{1}{r_i} + d_i \log \frac{m}{r_{\min}}\right)$. The edge queue size is bounded by the token lag $2L$, which is $O\left(\log \frac{m}{r_{\min}}\right)$. □

7.4 A Simple Distributed Scheduler

The above scheme is centralized since the session-i tokens on one template are dependent on the previous template. However, it suggests the following simple *distributed*

scheme for scheduling packets so as to achieve small delay. As in Section 7.3.2 we place initial tokens on the first edge of session i and delay each initial token by an amount chosen independently and uniformly at random from $[1, \ell_i]$, where ℓ_i is defined in Equation (7.1). (Note that the delay is from $[L+1, L+\ell_i]$ in the centralized scheme.) Suppose that a session-i packet p now obtains its initial token at time T. Then for the jth edge on the session-i path, p is given a deadline of $T+2L(j-1)+L$, where L is defined in (7.4). Whenever two or more packets contend for the same edge simultaneously. e packet with the earliest deadline moves. Ties are broken arbitrarily. We call this scheme EARLIEST-DEADLINE-FIRST (EDF). Note that EDF is no longer template based. We show in Lemma 7.4.1 that the deadlines do not cluster together with high probability, and show in Lemma 7.4.2 that every packet meets its deadlines.

Lemma 7.4.1 *For any edge, at most L deadlines appear in any consecutive L time steps with probability at least $1-1/(mM)$, where L is defined in (7.4) for a sufficiently large constant α.*

Proof: The deadlines for a packet p are $T+L$, $T+3L$, $T+5L$, ..., which correspond to the times that the tokens in a sequence appear. Hence, the proof is identical to that of Lemma 7.3.2. □

Lemma 7.4.2 *If for any edge at most L deadlines appear in any consecutive L time steps, then each packet crosses every edge by its deadline by* EDF.

Proof: For the purpose of contradiction, let D be the first deadline that is missed. This implies all deadlines earlier than D are met. Let p be the packet that misses deadline D for edge e. Since packet p makes its previous deadlines, p must have crossed its previous edge by time $D-2L$, or else e must be p's first edge and p must have obtained its initial token at time $D-L$. Hence, at every time step from time $D-L+1$ to D packet p is held up by another packet with deadline no later than D. Furthermore, these deadlines must be later than $D-L$ since all deadline earlier

than D are met. Therefore, at least $L+1$ packets have deadlines for edge e from time $D-L+1$ to D. Our lemma follows from the contradiction. □

By an argument similar to that in Theorem 7.3.4, a session-i packet obtains its initial token within $2\ell_i$ steps of its injection. Combined with Lemmas 7.4.1 and 7.4.2 we have,

Theorem 7.4.3 *With high probability, the randomized distributed scheme* EARLIEST-DEADLINE-FIRST *generates a schedule that produces an end-to-end delay bound of* $O\left(\frac{1}{r_i} + d_i \log \frac{m}{r_{\min}}\right)$.

Chapter 8

Outline of the Optimal Result

Our main result for the dynamic scheduling problem parallels an earlier result on static routing. In Section 8.1 we review techniques used for solving the static case, and in Section 8.2 we give an outline of the additional complexities that need to be addressed in the dynamic case. Relevant parameters are defined in Section 8.3.

8.1 A Bound of $O(c+d)$ for Static Routing

Leighton, Maggs and Rao consider the static routing problem for arbitrary networks in [31]. For static routing, all packets are present in the network initially. Each packet is associated with a source, a destination and a route. The *congestion* on each edge is the total number of routes that require this edge, and the *dilation* of a route is the number of edges on the route. Leighton et al. show that for any set of routes with maximum congestion c (over all edges) and maximum dilation d (over all routes), there is a schedule of length $O(c+d)$ and edge queue size $O(1)$. In this schedule, at most one packet traverses each edge at each time step. A packet waits $O(c+d)$ steps initially before leaving its source, and it waits $O(1)$ steps to cross each subsequent edge.

We summarize here the techniques in [31]. The strategy for constructing an efficient schedule is to make a succession of *refinements* to an initial greedy schedule. Let $\mathcal{S}^{(q)}$ represent the schedule in the qth iteration. In the initial schedule $\mathcal{S}^{(0)}$, each

packet moves at every step until it reaches its destination. This schedule has length d, but as many as c packets may traverse the same edge at the same step. Each refinement brings the schedule closer and closer to the requirement that at most one packet uses one edge per time step.

Let us introduce a few concepts here. A *T-frame* is a time interval of length T. The *frame congestion*, C, in a T-frame is the largest number of packets that use any edge during the frame. The *relative congestion* in a T-frame is the ratio C/T, where C is the frame congestion.

It is obvious that the initial schedule $\mathcal{S}^{(0)}$ has relative congestion at most 1 for any c-frame. A *refinement* transforms a schedule $\mathcal{S}^{(q)}$ with relative congestion at most $c^{(q)}$ for any frame of size $I^{(q)}$ or larger into a schedule $\mathcal{S}^{(q+1)}$ with relative congestion at most $c^{(q+1)}$ for any frame of size $I^{(q+1)}$ or larger. The resulting frame size $I^{(q+1)}$ is much smaller than $I^{(q)}$, whereas the relative congestion $c^{(q+1)}$ is only slightly bigger than $c^{(q)}$. In particular, $I^{(q+1)} = \log^5 I^{(q)}$ and $c^{(q+1)} = (1 + o(1))c^{(q)}$.

Schedule	Frame size	Relative congestion
$\mathcal{S}^{(q)}$	$I^{(q)}$	$c^{(q)}$
Refinement	$\log^5 I^{(q)}$	$(1 + o(1))c^{(q)}$
$\mathcal{S}^{(q+1)}$	$I^{(q+1)}$	$c^{(q+1)}$

Each refinement is achieved by inserting random delays to the packets. The intuition is that enough randomness would prevent many packets from crossing the same edge during any small time intervals. It is the central issue in [31] to show that the packets can always be delayed in such a way that the relative congestion in every $I^{(q+1)}$-frame is small.

After a series of $O(\log^* c)$ refinements, a schedule $\mathcal{S}^{(\zeta)}$ is obtained where the relative congestion is $O(1)$ for any $O(1)$-frame. A final schedule that satisfies the condition of one packet per edge per time step can be constructed by stretching $\mathcal{S}^{(\zeta)}$ by a constant factor.

8.2 A Bound of $O(1/r_i + d_i)$ for Dynamic Routing

Our result for the dynamic routing problem is parallel to that in [31]. For an arbitrary network where paths (sessions) are defined, we show that there is a schedule such that every session-i packet reaches its destination within $O(1/r_i + d_1)$ steps of its injection, where r_i and d_i are the injection rate and path length respectively for session i. A session-i packet waits $O(1/r_i + d_i)$ steps initially before leaving its source, and it waits $O(1)$ steps to cross each edge afterwards.

To achieve a session-based, end-to-end delay bound of $O(1/r_i + d_i)$ for our dynamic routing problem, we adopt the general approach in [31]. However, there are three major problems in transforming the solution for the static problem into a solution for the dynamic problem. In the following we present these three problems and their solutions.

Problem 1: Infinite Time

In [31] all the packets to be scheduled are present initially. In the dynamic model, packets are injected over an infinite time line. We would like to partition the infinite time line into finite time intervals that can be scheduled independently of each other. We divide time into intervals of length $\mathcal{T}$, where $\mathcal{T}$ is defined in Section 8.3. We shall independently schedule the time intervals $[0, \mathcal{T})$, $[\mathcal{T}, 2\mathcal{T})$, $[2\mathcal{T}, 3\mathcal{T})$, etc.

We associate each session i with a quantity $\mathcal{T}_i = O(1/r_i + d_i)$, which is also defined in Section 8.3. For any integer $k \geq 0$ consider all the session-i packets that are injected during interval $[k\mathcal{T} - \mathcal{T}_i, (k+1)\mathcal{T} - \mathcal{T}_i)$. We shall provide a schedule in which all these packets leave their sources no earlier than time $k\mathcal{T}$ and reach their destinations before time $(k+1)\mathcal{T}$. (See Figure 8-1.) From now on, we shall concentrate on scheduling the arrivals that would be serviced during interval $[\mathcal{T}, 2\mathcal{T})$.

Problem 2: Session-Based Delay Guarantees

Once we restrict ourselves to interval $[\mathcal{T}, 2\mathcal{T})$, it seems that dynamic routing problem is similar to static routing. However, we cannot simply proceed with the successive

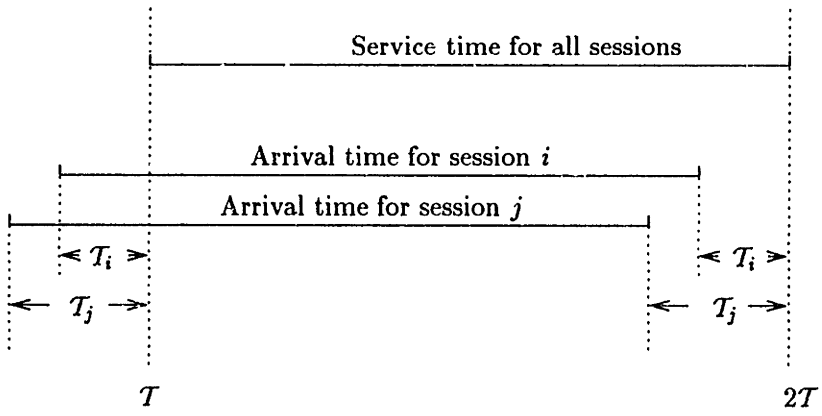


Figure 8-1: All the session-i packets that arrive during $[k\mathcal{T} - \mathcal{T}_i, (k+1)\mathcal{T} - \mathcal{T}_i)$ are serviced during $[k\mathcal{T}, (k+1)\mathcal{T})$. In this figure, $k = 1$.

refinements as in Section 8.1, since some sessions need tighter delay bounds than others. Session-i packets can only tolerate a delay proportional to $1/r_i + d_i$. Sessions are grouped according to their associated $1/r_i + d_i$ values. We start by inserting delays to sessions having large values of $1/r_i + d_i$, reducing the frame size and bounding the relative congestion. When the frame size becomes small enough, sessions with smaller $1/r_i + d_i$ join in.

More precisely, we introduce the concept of *integral* and *fractional* sessions. When session i is *integral*, packets of size 1 are injected at rate r_i. When session i is *fractional*, a packet of size $\hat{r}_i$ is injected at every time step, where $\hat{r}_i \approx r_i$ is defined in Section 8.3. A packet from a fractional session always crosses one edge at a time, whether or not other packets are crossing the edge at the same time. Therefore, a fractional packet from session i always contributes exactly $\hat{r}_i$ to the congestion. Integral sessions are those to which we can afford to insert delays in order to bound the congestion. Fractional sessions are those to which we cannot insert delays. However, congestion due to a fractional session i is only $\hat{r}_i$, which is small.

As before, $\mathcal{S}^{(q)}$ represents the schedule in the qth iteration. The set of integral sessions for $\mathcal{S}^{(q)}$ is denoted by $A^{(q)}$. For the initial schedule $\mathcal{S}^{(0)}$, all the sessions are fractional and we show that the relative congestion is less than 1. For schedule $\mathcal{S}^{(q)}$ we inductively assume that the relative congestion due to the current integral and fractional sessions is at most $c^{(q)}$ for any frame of size $I^{(q)}$ or larger. To create a

schedule $\mathcal{S}^{(q+1)}$ from schedule $\mathcal{S}^{(q)}$ we carry out a frame-refinement step in Section 9.2 and a conversion step in Section 9.3.

The frame-refinement step reduces the frame size from $I^{(q)}$ to $I^{(q+1)} = \log^5 I^{(q)}$, while slightly increasing the relative congestion from $c^{(q)}$ to $(1 + o(1))c^{(q)}$. This step is achieved by delaying the integral packets by up to $\Theta\left((I^{(q)})^2\right)$ steps. We make sure that if session i is in $A^{(q)}$ then $1/r_i + d_i \geq (I^{(q)})^2$, and therefore the delays inserted can be tolerated. The conversion step converts some sessions from fractional to integral, while maintaining the frame size of $I^{(q+1)}$ and slightly increasing the relative congestion to $c^{(q+1)} = (1 + o(1))^2 c^{(q)}$. These newly-converted sessions form a set $B^{(q+1)}$ and have associated values $1/r_i + d_i \geq (I^{(q+1)})^2$. This bound is chosen so that the sessions in $A^{(q+1)}$, which is $A^{(q)} \cup B^{(q+1)}$, will be able to tolerate the delays inserted during the next iteration of frame-refinement. During the conversion step we delay the packets in $B^{(q+1)}$ by up to $\Theta(1/r_i + d_i)$ steps. We shall show the existence of "good" delays for both frame-refinement and conversion steps. The following table summarizes our approach.

Schedule	Integral sessions	Frame size	Relative congestion
$\mathcal{S}^{(q)}$	$A^{(q)}$	$I^{(q)}$	$c^{(q)}$
Refinement	$A^{(q)}$	$\log^5 I^{(q)}$	$(1 + o(1))c^{(q)}$
Conversion	$A^{(q)} \cup B^{(q+1)}$	$\log^5 I^{(q)}$	$(1 + o(1))^2 c^{(q)}$
$\mathcal{S}^{(q+1)}$	$A^{(q+1)}$	$I^{(q+1)}$	$c^{(q+1)}$

In Section 9.4 we show that after a succession of refinement and conversion we have a schedule $\mathcal{S}^{(\zeta)}$ in which every session is integral and the relative congestion is at most 1 for all frames of size larger than a certain constant. In $\mathcal{S}^{(\zeta)}$ all session-i arrivals during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ are serviced during $[\mathcal{T}, 2\mathcal{T})$. Furthermore, all session-i packets reach their destination within $O(\mathcal{T}_i)$ steps of their injections.

Problem 3: Constant-Factor Stretching in the Final Schedule

We now have a schedule $\mathcal{S}^{(\zeta)}$ in which all sessions are integral and in which the relative congestion is 1 for any frames of size larger than a certain constant. In the static

problem, a final schedule can easily be obtained by stretching $\mathcal{S}^{(\zeta)}$ by a constant factor. However, we cannot afford to have a constant blowup in our final schedule for the dynamic routing. This is because we need to independently schedule all time intervals $[0, \mathcal{T})$, $[\mathcal{T}, 2\mathcal{T})$, etc, and a constant blowup would make these time intervals overlap.

To overcome this problem, we shall first devise a schedule for an intermediate network $\mathcal{M}$. We construct $\mathcal{M}$ from the original network $\mathcal{N}$ as follows. Each edge e of $\mathcal{N}$ is replaced by $2w$ consecutive edges $e_1, \ldots, e_{2w}$, where w is a constant defined in Section 8.3. The rates and routes of the sessions are unaffected. In $\mathcal{M}$, session i has length $D_i = 2wd_i = O(d_i)$. All the techniques described earlier are applied to the network $\mathcal{M}$. We carry out successive of refinement and conversion steps for $\mathcal{M}$ and obtain a schedule $\mathcal{S}^{(\zeta)}$, where the relative congestion is 1 for any frame whose size is larger than w. We then "smooth" $\mathcal{S}^{(\zeta)}$ and convert it to a schedule for $\mathcal{N}$ where only one packet at a time traverses any link.

The idea behind the smoothing process is as follows. In $\mathcal{S}^{(\zeta)}$, there may be more than 1 packet which requires some edge of $\mathcal{M}$ during a given time step. However, at most w packets will require any given edge in $\mathcal{M}$ within w time steps. This fact means we can shuffle in time the packets that require the edge, so that exactly one packet traverses the edge at any time step. Unfortunately, this shuffling in time can lead to an impossible schedule for $\mathcal{M}$, in that a packet is assigned to traverse its edges out of order (timewise). However, it turns out that if one only considers the schedule with respect to the packets traversing edge e_{2w}, for all e, then the schedule is completely legal, i.e. the packets cross *these* edges in order. The idea is then to schedule edge e in $\mathcal{N}$ in the same way that the corresponding edge e_{2w} is being scheduled in $\mathcal{M}$. We shall explain it in detail in Section 9.5.

Figure 8-2 is a schematic picture of our overall approach.

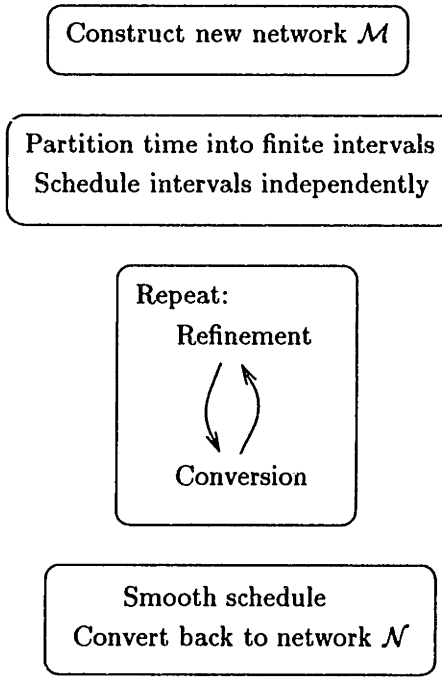


Figure 8-2: An overview of our approach for the dynamic routing problem.

8.3 Parameter Definitions

Interval Lengths

To resolve Problem 1 of Section 8.2 we independently schedule intervals $[0, \mathcal{T})$, $[\mathcal{T}, 2\mathcal{T})$, etc. The parameter $\mathcal{T}$ also serves as the template size. Our proof will concentrate on the interval $[\mathcal{T}, 2\mathcal{T})$. All the session-$i$ packets that arrive during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ are serviced during $[\mathcal{T}, 2\mathcal{T})$. We define $\mathcal{T}$ and $\mathcal{T}_i$ for session i as follows. Recall $D_i = 2wd_i$, where w is a constant defined at the end of this chapter.

$$\begin{aligned} \mathcal{T}_i &= 4D_i + 1 + (8/\varepsilon + 2)/r_i, \\ \mathcal{T} &= \left\lceil \frac{(1 + 4/\varepsilon)\max_i \mathcal{T}_i}{w} \right\rceil w. \end{aligned}$$

In other words, $\mathcal{T}$ is the smallest multiple of w that is greater than or equal to $(1 + 4/\varepsilon)\max_i \mathcal{T}_i$.

It is easy to see that $\mathcal{T}_i = O(1/r_i + D_i) = O(1/r_i + d_i)$. Our choices for $\mathcal{T}$ and $\mathcal{T}_i$ are justified in Theorem 9.4.4, Lemma 9.3.1 and the smoothing process of Section 9.5.

Packet Size for Fractional Sessions

To resolve Problem 2 of Section 8.2, we maintain fractional sessions and integral sessions. A fractional session-i injects a packet of size $\hat{r}_i$ at every step. An integral session injects packets of size 1 at rate r_i.

We define $\hat{r}_i$ to be s_i/ℓ_i, a fraction slightly larger than r_i. In particular,

$$\ell_i = \lceil 8/(\varepsilon r_i) \rceil \tag{8.1}$$

$$s_i = \lfloor \ell_i r_i (1 + \varepsilon/2) \rfloor, \tag{8.2}$$

where ε is such that the total injection rate on any edge is at most $1-\varepsilon$. An argument similar to Inequality 7.3 of Lemma 7.3.1 implies,

$$r_i(1 + \varepsilon/4) + 1/\ell_i \leq \hat{r}_i \leq r_i(1 + \varepsilon/2). \tag{8.3}$$

Note that the definition of ℓ_i and the left-hand side of the above inequality are different from the ones in Section 7.3.1. We need this stronger lower bound on $\hat{r}_i$ to handle the extra complexity in the conversion step. In particular, $\hat{r}_i$ is also used to indicate rate at which the initial tokens for session-i appear. During the conversion step, the initial tokens for session-i are placed in the interval $[\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i)$. Since these tokens are to accommodate all the session-i arrivals during $[\mathcal{T}, 2\mathcal{T})$, we need $\hat{r}_i(\mathcal{T} - \mathcal{T}_i) \geq r_i\mathcal{T}$. This condition is guaranteed by the choices of $\mathcal{T}$ and $\mathcal{T}_i$ and Inequality (8.3). (See Lemma 9.3.1.)

Since $\varepsilon \in (0, 1)$ and the total injection rate on any edge is at most $1 - \varepsilon$, the right-hand side of Inequality (8.3) implies,

Lemma 8.3.1 *For all edges e, $\sum_{i \in S_e} \hat{r}_i < 1 - \varepsilon/2$ where S_e is the set of sessions that use edge e.*

Parameters for Schedule $\mathcal{S}^{(q)}$

We define relevant parameters for the frame-refinement and conversion steps for schedule $\mathcal{S}^{(q)}$. For $\mathcal{S}^{(q)}$, $A^{(q)}$ consists of all the integral sessions. The relative congestion,

due to all integral and fractional sessions, is at most $c^{(q)}$ for any frame of size $I^{(q)}$ or larger. As we construct schedule $\mathcal{S}^{(q+1)}$, sessions in $B^{(q+1)}$ become integral and join $A^{(q)}$. The succession of refinement and conversion terminates at schedule $\mathcal{S}^{(\zeta)}$, when $I^{(\zeta)}$ is less than or equal to a constant w defined in the next section.

The parameters $I^{(q)}$, $c^{(q)}$, $A^{(q)}$ and $B^{(q+1)}$ are defined by the following recurrences. Let $X_i = D_i + 1/r_i$ for session i, and let $X_{\max} = \max_i X_i$.

$$
\begin{aligned}
I^{(0)} &= e^{\log^{2/5} X_{\max}} \\
I^{(q+1)} &= \log^5 I^{(q)} \\
c^{(0)} &= 1 - \varepsilon/2 \\
c^{(q+1)} &= (1 + \delta^{(q)})^2 c^{(q)} \\
\delta^{(q)} &= \frac{\alpha}{\sqrt{\log I^{(q)}}}, \qquad \text{for a sufficiently large constant } \alpha \\
A^{(0)} &= \emptyset \\
A^{(q+1)} &= A^{(q)} \cup B^{(q+1)} \\
B^{(q+1)} &= \left\{ i \notin A^{(q)} : \left(I^{(q+1)}\right)^2 \le X_i \le e^{\sqrt{I^{(q+1)}}} \right\} \qquad \text{for } q \neq \zeta - 1 \\
B^{(q+1)} &= \left\{ i \notin A^{(q)} : X_i \le e^{\sqrt{I^{(q+1)}}} \right\} \qquad \text{for } q = \zeta - 1
\end{aligned}
$$

The above parameter definitions comply with our discussion of Problem 2 in Section 8.2. During each iteration, the frame size decreases polylogarithmically and the relative congestion increases by a small factor of $1+o(1)$. Sessions with large values of X_i become integral first. In the definition of $B^{(q+1)}$, we use the bound $\left(I^{(q+1)}\right)^2 \le X_i$ in the frame-refinement step and we use the bound $X_i \le e^{\sqrt{I^{(q+1)}}}$ in the conversion step.

Definition of w

We define a constant w that has two purposes. First, the process of refinement and conversion terminates when the frame size becomes smaller than or equal to w.

Second, the intermediate network $\mathcal{M}$ is constructed from the original network $\mathcal{N}$ by replacing each edge in $\mathcal{N}$ with $2w$ edges. We define w to be a constant that satisfies the following two bounds.

1. $w \geq x$, where x satisfies $\left(1 - \frac{\alpha}{\sqrt{\log x}}\right)^2 = 1 - \varepsilon/2$, i.e. $x = e^{\alpha^2(1-\sqrt{1-\varepsilon/2})^{-2}}$;
2. $w \geq 2\log^{15} w + 2\log^{10} w - \log^5 w$.

The first bound ensures that the relative congestion $c^{(\zeta)}$ is at most 1. (See Lemma 9.4.2.) The second bound is to maintain an invariant throughout the frame-refinement steps. (See Section 9.2.) The second bound also implies the following.

3. $w \geq e^{2^{2/3}}$;
4. $w \geq 4\log^5 w$, which implies $w^2 + w \geq 4(\log^{10} w + \log^5 w)$.

The third bound is to guarantee that the $B^{(q)}$'s form a partition of all the sessions. (See Lemma 9.4.1.) The fourth bound is to upper bound the total delay inserted during the frame-refinement step. (See Lemma 9.4.3.)

Chapter 9

The Existence of an Optimal Schedule

In this chapter we show the existence of an asymptotically-optimal schedule. Sections 9.1 through 9.4 concentrate on Problem 2 of Section 8.2. We begin with an initial schedule $\mathcal{S}^{(0)}$ and transform it to schedule $\mathcal{S}^{(\zeta)}$ through a process of refinement and conversion. All these schedules are designed for the intermediate network $\mathcal{M}$. Section 9.5 concentrates on Problem 3 of Section 8.2. We describe how to obtain an optimal schedule $\mathcal{S}_{\mathcal{N}}$ for the original network $\mathcal{N}$ from $\mathcal{S}^{(\zeta)}$.

9.1 An Initial Schedule $\mathcal{S}^{(0)}$

In $\mathcal{S}^{(0)}$, all sessions are fractional, i.e. $A^{(0)} = \emptyset$. Each packet (of a fractional size) crosses one edge per time step whether or not other packets are using the same edge at the same time. Since the relative congestion is entirely due to fractional sessions, the relative congestion is at most $\sum r_i < 1 - \varepsilon/2 = c^{(0)}$ on any edge e due to Lemma 8.3.1.

Note that the above relative congestion holds for any frame size. We choose the initial frame size $I^{(0)} = e^{\log^{2/5} X_{\max}}$, so that $I^{(1)} = \log^2 X_{\max}$. The definition of $B^{(1)}$ allows the sessions with the largest X_i values to be converted during the first iteration of the conversion process.

9.2 Frame-Refinement for Schedule $\mathcal{S}^{(q)}$

In this section we describe the frame-refinement process. During this process, random delays are inserted to integral packets in order to reduce the frame size without increasing the relative congestion by much. For schedule $\mathcal{S}^{(q)}$, we inductively assume that the relative congestion is at most $c^{(q)}$ for frames of size $I^{(q)}$ or larger. We show that there is a way to delay the integral packets from $A^{(q)}$ so that, in the resulting schedule $\mathcal{S}^{(q+\frac{1}{2})}$, the relative congestion is at most $(1+\delta^{(q)})c^{(q)}$ for any frame of size $I^{(q+1)} = \log^5 I^{(q)}$ or larger, where $\delta^{(q)} = \frac{\alpha}{\sqrt{\log I^{(q)}}}$. Throughout the construction we also maintain the following invariant.

[Invariant] *For any $q \geq 0$, every integral packet waits at most once every $I^{(q)}$ steps after leaving its source according to schedules $\mathcal{S}^{(q+\frac{1}{2})}$ and $\mathcal{S}^{(q+1)}$.*

The base case of the initial schedule $\mathcal{S}^{(0)}$ is described in Section 9.1. Since there are no integral sessions, no delays are inserted in this step. Trivially, the resulting relative congestion is at most $(1+\delta^{(0)})c^{(0)}$ for any frame of size $I^{(1)}$ or larger at the end of this step, and no packet ever waits. The invariant is also maintained.

Let us now consider refining schedule $\mathcal{S}^{(q)}$, for $q > 0$. Frame refinement is divided into two steps. In the first refinement step we divide the current schedule into blocks of length $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}$, and insert delays into each block so that its length increases to $2(I^{(q)})^3 + 2(I^{(q)})^2$. We show that these delays can be introduced in such a way that in the central $2(I^{(q)})^3$ steps of each block the relative congestion of frames of at least length $I^{(q+1)}$ is only a little larger than $c^{(q)}$. (See Figure 9-1.) At the beginning and end of each block there are "fuzzy" regions of length $(I^{(q)})^2$ each. In the second step we move the block boundaries so that the fuzzy regions are at the center of the new blocks of $2(I^{(q)})^3 + 2(I^{(q)})^2$ steps. (See Figure 9-2.) Again, we insert delays into each block increasing the size of the block by $(I^{(q)})^2$ steps. We show that there is a way to insert these delays so that the final conditions for refining $\mathcal{S}^{(q)}$ are indeed satisfied. (See Figure 9-3.)

Before we present the two steps of refinement in Sections 9.2.2 and 9.2.3, we first prove Lemma 9.2.2 that will be used extensively in both steps of the refinement.

9.2.1 A Useful Lemma

In the following we present Lemma 9.2.2, which is based on Lemma 9.2.1.

Lemma 9.2.1 *Let X and Y be independent random variables. Let Y be binomially distributed with mean μ_y, and let σ_1, σ_2, and v be values such that $\sigma_2 = \sigma_1 - 1/v$. Then,*

$$\Pr[\, X + \mu_y > (1+\sigma_1)v \,] \leq 2\Pr[\, X + Y > (1+\sigma_2)v \,].$$

Proof: Let $w = (1+\sigma_1)v - \mu_y$. We have,

$$\Pr[\, X + \mu_y > (1+\sigma_1)v \,] = \Pr[\, X > w \,], \tag{9.1}$$

$$\Pr[\, X + Y > (1+\sigma_2)v \,] = \Pr[\, X + Y > \mu_y + w - 1 \,]. \tag{9.2}$$

Note also that,

$$\begin{aligned}\Pr[\, X + Y > \mu_y + w - 1 \,] &\geq \Pr[\, X > \mu_y + w - 1 - \lfloor \mu_y \rfloor \text{ and } Y \geq \lfloor \mu_y \rfloor \,] \\ &= \Pr[\, X > w - 1 + \mu_y - \lfloor \mu_y \rfloor \,] \Pr[\, Y \geq \lfloor \mu_y \rfloor \,].\end{aligned}$$

This last equality follows from the independence of X and Y. Theorem B.1 in [33] shows that $\Pr[\, Y \geq \lfloor \mu_y \rfloor \,] \geq 1/2$. Since $\mu_y - \lfloor \mu_y \rfloor < 1$, we have,

$$\Pr[\, X + Y > \mu_y + w - 1 \,] \geq \frac{1}{2}\Pr[\, X > w \,].$$

Our Lemma follows from Equalities (9.1) and (9.2) and the above inequality. □

Given some schedule, a *region* R is some interval of contiguous time steps in this schedule. A packet is *active* during some region if the packet belongs to some integral session and it traverses at least one edge during the region. Throughout the construction, the invariant we maintain implies that an *inactive* packet is either at its source or its destination during the entire region (as long as the region consists of at least two time steps). Lemma 9.2.2 below is a stepping stone that allows us to reduce the frame size from $I^{(q)}$ to *poly* $\log I^{(q)}$. We invoke this lemma for various values of s, t, r and R.

Lemma 9.2.2 *Consider some region R of a schedule where the relative congestion is at most $r = \Theta(1)$ for frames of length s, where $\log^3 I^{(q)} \leq s \leq (I^{(q)})^2$. Consider any edge e and any t-frame, where $\log^2 I^{(q)} \leq t \leq 2\log^2 I^{(q)}$. Assume each active packet in the region is delayed between the beginning of R and the beginning of the t-frame by a number of steps randomly, independently, and uniformly chosen from $[1, s]$. Then, for any constant k there is some value $\gamma = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$ such that the probability of having a relative congestion larger than $r(1+\gamma)$ on e during the t-frame is at most $(I^{(q)})^{-k}$.*

Proof: Let the random variable X be the frame congestion on e during the t-frame due to the active packets after they are delayed. If the relative congestion due to fractional sessions is r_f, the frame congestion due to fractional sessions in the t-frame is exactly $r_f t$. Since the active packets are the only integral-session packets that can cross e during the region, the frame congestion on e during the t-frame is $X + r_f t$ after the delay.

Let Y be a binomial random variable with parameters $(r_f s, t/s)$ and mean $E[Y] = r_f t$. From Lemma 9.2.1, the probability p that the congestion in the t-frame is larger than $(1+\gamma)rt$ after the packets have been delayed is,

$$p = \Pr[\, X + r_f t > (1+\gamma)rt \,] \leq 2\Pr[\, X + Y > (1+\sigma)rt \,],$$

where $\sigma = \gamma - 1/rt$. Since $t \geq \log^2 I^{(q)}$ and $r = \Theta(1)$, then $\gamma = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$ if and only if $\sigma = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$. Let $\sigma = \frac{v}{\sqrt{\log I^{(q)}}}$, where v is a constant. We shall choose an appropriate value v so that the lemma is satisfied.

To bound probability p we show that X is a binomial random variable with parameters (m, ρ), where $m \leq (t+s)(r - r_f)$ and $\rho \leq t/s$. Since the active packets are delayed up to s steps, an active packet that crosses e in the t-frame after the delay could only cross e in an interval of $t+s$ steps before the delay. The relative congestion due to active packets is at most $r - r_f$ in that interval before the delay. Hence, at most $(t+s)(r - r_f)$ active packets can cross e in the t-frame after the delay. Each of these packet independently does so with a probability at most t/s, since delays are chosen independently and uniformly from $[1, s]$ and all session paths are simple.

Recall that Y is a binomial random variable with parameters $(r_f s, t/s)$. We define Z to be a binomial random variable with parameters $(n, t/s)$, where $n = r(t+s) > (r - r_f)(t+s) + r_f s$. It is easy to see that,

$$p \leq 2\Pr[\, X + Y > (1+\sigma)rt \,] \leq 2\Pr[\, Z > (1+\sigma)rt \,].$$

We use a Chernoff bound to bound p. Since $E[Z] = (t+s)rt/s$, we have,

$$\Pr[\, Z > (1+\sigma)rt \,] = \Pr\left[\, Z > \left(1 + \tfrac{\sigma - t/s}{1+t/s}\right) E[Z] \,\right] \leq e^{-\frac{rt}{3}(1+t/s)\left(\frac{\sigma - t/s}{1+t/s}\right)^2}.$$

Since $t/s = o(\sigma)$ and $t/s = o(1)$ by the bounds on s and t, the above probability is bounded by $e^{-\Theta(rt\sigma^2)}$. Hence,

$$p \leq 2e^{-\Theta(rt\sigma^2)}.$$

The bound on t and the definitions of r and σ imply that we can choose a constant v large enough so that $p < (I^{(q)})^{-k}$ for any constant $k > 0$. □

9.2.2 The First Refinement Step for $\mathcal{S}^{(q)}$

Suppose the current schedule $\mathcal{S}^{(q)}$ schedules every session-i packet injected during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ to move during $[\mathcal{T}, \mathcal{T} + x)$, where $0 < x < \mathcal{T}$. We first partition the region $[\mathcal{T}, \mathcal{T} + x)$ into consecutive blocks of length $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}$, and reschedule each block independently.

Within each block B only active packets are delayed. In particular, each active packet in B is assigned a delay randomly, uniformly and independently chosen from $[1, I^{(q)}]$. An active packet p, whose assigned delay is x, is delayed in the first $xI^{(q)}$ steps of B once every $I^{(q)}$ steps. In order to independently reschedule the next block, packet p is also delayed in the last $(I^{(q)} - x)I^{(q)}$ steps of B once every $I^{(q)}$ steps. Therefore, a rescheduled block has length $2(I^{(q)})^3 + 2(I^{(q)})^2$.

Before the delays are inserted to reschedule block B, an active packet p is delayed at most once within the block, provided that $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)} < I^{(q-1)}$, which holds as long as $I^{(q-1)}$ is larger than w, the constant defined in Section 8.3. (See the

second bound in the definition of w.) Prior to inserting any new delay to a block, we check if it is within $I^{(q)}$ steps of the single old delay. If the new delay would be too close to the old delay, then it is simply not inserted. The loss of one delay in a block has a negligible effect on the probability analysis that follows. In this way, we maintain the invariant.

Lemma 9.2.4 shows that with the insertion of delays we can dramatically reduce the frame size in the center of the block and increase the relative congestion only slightly. The following fact is used in Lemma 9.2.4 and on a few other occasions.

Lemma 9.2.3 *If the relative congestion in every frame of size T to $2T-1$ is at most r, then the relative congestion in any frame of size T or greater is at most r.*

Proof: Consider a frame of size T', where $T' > 2T-1$. The first $\lfloor T'/T \rfloor T - T$ steps of the frame can be broken into T-frames, each with relative congestion r. The remainder of the T'-frame consists of a single frame of size between T and $2T-1$ steps in which the relative congestion is also at most r. □

Lemma 9.2.4 *There exists a way of choosing delays so that in between the first and last $(I^{(q)})^2$ steps of the block B, the relative congestion of any frame of size $\log^2 I^{(q)}$ or larger is at most $(1+\gamma_1)c^{(q)}$, for some $\gamma_1 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$.*

Proof: Due to Lemma 9.2.3, it is sufficient to prove the statement for all frames of size between $\log^2 I^{(q)}$ and $2\log^2 I^{(q)}$. We associate a bad event with each edge e and each I-frame, where $\log^2 I^{(q)} \leq I < 2\log^2 I^{(q)}$ and the I-frame lies entirely between the first and last $(I^{(q)})^2$ steps of B. A bad event $E_{\{e,I\}}$ happens when the frame congestion on edge e is more than $(1+\gamma_1)c^{(q)}I$ during the I-frame. We use the Lovász Local Lemma to show that with a nonzero probability no bad event occurs.

We first bound the dependence, d, of bad events. The probability space is given by the delays assigned to packets from sessions in $A^{(q)}$. Hence, a bad event $E_{\{e,I\}}$ is dependent on another bad event $E_{\{e',I'\}}$ only if an integral packet can cross edges e and e' in B. In block B, at most $c^{(q)}\left(2(I^{(q)})^3 + 2(I^{(q)})^2\right)$ integral packets cross the same edge, and each packet crosses at most $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}$ edges. The

number of I-frames in B is at most $\log^2 I^{(q)} \left(2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}\right)$. Hence, the dependency d is upper bounded by the product of these three quantities above. Since $c^{(q)} \leq 1$, d is $O\left(\log^2 I^{(q)}(I^{(q)})^9\right)$.

We now bound the probability, p, that one particular bad event $E_{\{e,I\}}$ happens, for some I-frame between the first and last $(I^{(q)})^2$ steps of B and for $\log^2 I^{(q)} \leq I \leq 2\log^2 I^{(q)}$. By setting $R = B$, $r = c^{(q)}$, $s = I^{(q)}$ and $t = I$, we apply Lemma 9.2.2 to show that for any constant k_1 there is some value $\gamma_1 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$ such that the probability p of one particular bad event happening is smaller than $(I^{(q)})^{-k_1}$. In particular, we choose $k_1 = 10$.

Therefore, we have $4pd < 1$ and our lemma follows from the Lovász Local Lemma.

□

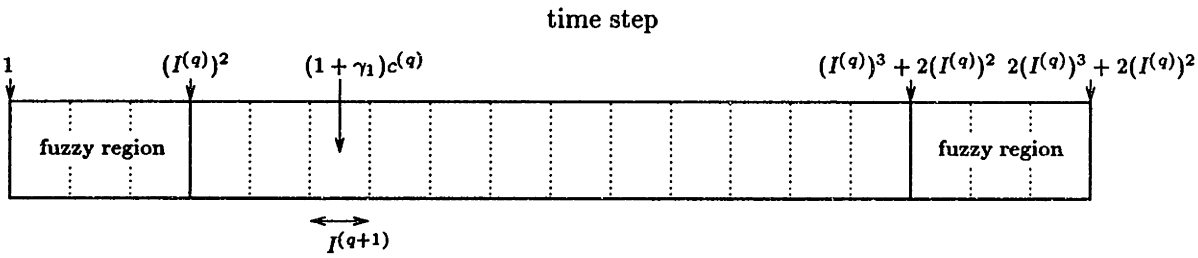


Figure 9-1: Situation after the first refinement step

At the end of the first refinement step, the center of each block has small relative congestion for small frame sizes. However, there are regions of $(I^{(q)})^2$ steps at the beginning and end of each block that may have very large relative congestion. We call these "fuzzy" regions, and we deal with them in the second refinement step.

9.2.3 The Second Refinement Step for $\mathcal{S}^{(q)}$

We start the second step of the refinement by relocating the block boundaries so that blocks still have $2(I^{(q)})^3 + 2(I^{(q)})^2$ steps, but now the fuzzy regions that were at the beginning and end of adjacent blocks are in the center of new blocks. Then, a new block has two "clean" regions of $(I^{(q)})^3$ steps each at the beginning and the end, and a fuzzy region of length $2(I^{(q)})^2$ steps in the center. (See Figure 9-2.)

As in the first step of the refinement we now concentrate on individual blocks.

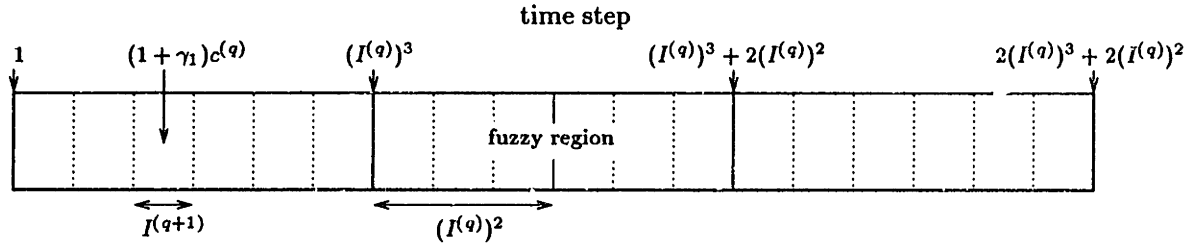


Figure 9-2: Situation after relocating block boundaries

We first show that the relative congestion is not very large for frames of size $(I^{(q)})^2$ or larger even in the fuzzy region.

Lemma 9.2.5 *For any choice of delays in the first step of the refinement, the relative congestion in any frame of size* $(I^{(q)})^2$ *or larger is at most* $(1 + 2/I^{(q)})c^{(q)}$.

Proof: Consider any edge e and any I-frame of size $(I^{(q)})^2$ or larger. Suppose I_1 steps of this I-frame appear before the center of the block and $I_2 = I - I_1$ steps after the center. (Either I_1 or I_2 can be zero). Each integral packet is delay by at most $I^{(q)}$ steps and each fractional packet is not delayed. Hence, a packet crosses edge e in the I_1-frame only if it did so in some frame of length $I_1 + I^{(q)}$ before the delays were inserted. Therefore, the frame congestion in the I_1-frame on any edge e at most $(I_1 + I^{(q)})c^{(q)}$. Similarly, the frame congestion on e in the I_2-frame is at most $(I_2 + I^{(q)})c^{(q)}$. Therefore, the frame congestion in the I-frame is at most $(I_1 + I_2 + 2I^{(q)})c^{(q)} = (I + 2I^{(q)})c^{(q)}$. For $I \geq (I^{(q)})^2$ the relative congestion is at most $(1 + 2/I^{(q)})c^{(q)}$. □

In order to reduce the frame size in the fuzzy region, we insert a random delay chosen independently and uniformly from $[1, (I^{(q)})^2]$ to each active packet in the block B. A packet p with delay x waits once every $(I^{(q)})^3/x$ at the beginning of the block and once every $(I^{(q)})^3/((I^{(q)})^2 - x)$ at the end. As in the first step a delay is not inserted if it is going to be within $I^{(q)}$ steps of an existing delay for an active packet. The block length after the delay insertion is $2(I^{(q)})^3 + 3(I^{(q)})^2$, and the fuzzy region can be $(I^{(q)})^2$ steps longer, spanning steps $(I^{(q)})^3$ to $(I^{(q)})^3 + 3(I^{(q)})^2$.

The following lemma shows that there is some way of inserting delays so that the frame size in the fuzzy region is reduced, and the frame size and relative congestion

in the rest of the block are increased by only a small amount. Figure 9-3 summarizes the final bounds on the frame size and the relative congestion.

Lemma 9.2.6 *In a block B, there exists a way of choosing delays so that in the fuzzy region (i.e. region $[(I^{(q)})^3, (I^{(q)})^3 + 3(I^{(q)})^2]$) the relative congestion of any frame of size $\log^2 I^{(q)}$ or larger is at most $(1+\gamma_2)c^{(q)}$, for some $\gamma_2 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$, and so that in the regions $[I^{(q)} \log^3 I^{(q)}, (I^{(q)})^3]$ and $[(I^{(q)})^3 + 3(I^{(q)})^2, 2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}]$ the congestion of any frame of size $\log^2 I^{(q)}$ or larger is at most $(1+\gamma_3)c^{(q)}$, for some $\gamma_3 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$.*

Proof: Due to Lemma 9.2.3 it is sufficient to prove the statement for all frames with size between $\log^2 I^{(q)}$ and $2\log^2 I^{(q)}$. As in Lemma 9.2.4, we use the Lovász Local Lemma. We associate a bad event with every edge e and every I-frame, for $\log^2 I^{(q)} \leq I \leq 2\log^2 I^{(q)}$. A bad event $E_{\{e,I\}}$ occurs, if

- more than $(1+\gamma_2)c^{(q)}I$ packets cross e in the I-frame in the fuzzy region $[(I^{(q)})^3, (I^{(q)})^3 + 3(I^{(q)})^2]$, or
- more than $(1+\gamma_3)c^{(q)}I$ packets cross e in the I-frame in the region $[I^{(q)} \log^3 I^{(q)}, (I^{(q)})^3]$ or in the region $[(I^{(q)})^3 + 3(I^{(q)})^2, 2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}]$.

The dependency, d, of the bad events is bounded as in Lemma 9.2.4. Two bad events are dependent if packets from some session $i \in A^{(q)}$ cross both edges in the block B. In B, $O\left((I^{(q)})^3\right)$ integral packets cross any edge, each of them crosses $O\left((I^{(q)})^3\right)$ other edges, and there are $O\left(\log^2 I^{(q)}(I^{(q)})^3\right)$ I-frames. Therefore, $d = O\left(\log^2 I^{(q)}(I^{(q)})^9\right)$.

To bound the probability p of one particular bad event happening, we consider the three regions separately and sum their respective probabilities.

We first consider the I-frames in the fuzzy region $[(I^{(q)})^3, (I^{(q)})^3 + 3(I^{(q)})^2]$. By Lemma 9.2.5, the relative congestion for frames of size $(I^{(q)})^2$ or longer is at most $(1+2/I^{(q)})c^{(q)} = \Theta(1)$. By choosing $R = B$, $r = (1+2/I^{(q)})c^{(q)}$, $s = (I^{(q)})^2$, and $t = I$, we use Lemma 9.2.2 to show that, for any constant k_2, there is some $\sigma_2 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$

such that the probability of having relative congestion on e in the I-frame larger than $c^{(q)}(1+2/I^{(q)})(1+\sigma_2) = c^{(q)}(1+\gamma_2)$ is smaller than $(I^{(q)})^{-k_2}$. Note that $\gamma_2 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$.

We now consider the I-frames in the region of $[I^{(q)} \log^3 I^{(q)}, (I^{(q)})^3]$. Suppose an I-frame starts at time step j. Given the way delays are inserted, by the jth step an active packet with delay x has been delayed $jx/(I^{(q)})^3$ steps. Thus, the delay of an active packet at the jth step is essentially a random value uniformly chosen from $[1, j/I^{(q)}]$. For $j \geq I^{(q)} \log^3 I^{(q)}$, we have $j/I^{(q)} \geq \log^3 I^{(q)}$. Note that before inserting delays, from Lemma 9.2.4 the relative congestion in any frame of length $\log^2 I^{(q)}$ or larger in the interval $[1, (I^{(q)})^3]$ was at most $(1+\gamma_1)c^{(q)}$. By choosing $R = [1, (I^{(q)})^3]$, $r = (1+\gamma_1)c^{(q)}$, $s = \log^3 I^{(q)}$, and $t = I$, we use Lemma 9.2.2 to show that, for any constant k_3, the existence of some $\sigma_3 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$ such that the probability of having relative congestion larger than $(1+\sigma_3)(1+\gamma_1)c^{(q)} = (1+\gamma_3)c^{(q)}$ on e in the I-frame is smaller than $(I^{(q)})^{-k_3}$. Again, $\gamma_3 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$.

By symmetry, the same value γ_3 makes the probability of a bad event happening on e in a I-frame in $[(I^{(q)})^3 + 3(I^{(q)})^2, 2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}]$ smaller than $(I^{(q)})^{-k_3}$.

We choose values for k_2 and k_3 such that the probability of one particular bad event is bounded as $p = O((I^{(q)})^{10})$. Therefore, we guarantee $4pd < 1$ and our lemma follows from the Lovász Local Lemma. □

Finally, we bound the frame size and the relative congestion in the remaining regions of the block in the following lemma.

Lemma 9.2.7 *The relative congestion in any frame of size* $\log^4 I^{(q)}$ *or larger in the regions* $[1, I^{(q)} \log^3 I^{(q)}]$ *and* $[2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}, 2(I^{(q)})^3 + 3(I^{(q)})^2]$ *is at most* $(1+\gamma_1)(1+1/\log I^{(q)})c^{(q)} = (1+\gamma_4)c^{(q)}$, *where* $\gamma_4 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$.

Proof: Let us first consider some I-frame in the region $[1, I^{(q)} \log^3 I^{(q)}]$. Recall that, before inserting delays, the relative congestion for frames of size $\log^2 I^{(q)}$ or more was at most $(1+\gamma_1)c^{(q)}$. In this region no packet is delayed more than $\log^3 I^{(q)}$ steps. Therefore, a packet crosses edge e in the I-frame only if it did so in some $(I + \log^3 I^{(q)})$-frame before the delay were inserted. The frame congestion for this

I-frame is therefore at most $(I + \log^3 I^{(q)})(1 + \gamma_1)c^{(q)}$. For $I \geq \log^4 I^{(q)}$ our claim follows. The proof for region $[2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}, 2(I^{(q)})^3 + 3(I^{(q)})^2]$ is similar. □

From Lemmas 9.2.6 and 9.2.7 we conclude that any frame of size at least $\log^4 I^{(q)}$ in each of the different regions has relative congestion at most $(1 + \gamma)c^{(q)}$, where $\gamma = \max(\gamma_2, \gamma_3, \gamma_4)$ and $\gamma = \frac{O(1)}{\sqrt{\log I^{(q)}}}$. We need to be careful with the relative congestion in frames that overlap several regions or several blocks. We can safely say that for any frame of size $I^{(q+1)} = \log^5 I^{(q)}$ or larger in the schedule $\mathcal{S}^{(q+\frac{1}{2})}$ obtained after the frame-refinement, the relative congestion is at most $(1 + \delta^{(q)})c^{(q)}$, for some $\delta^{(q)} = \frac{\alpha}{\sqrt{\log I^{(q)}}}$ large enough.

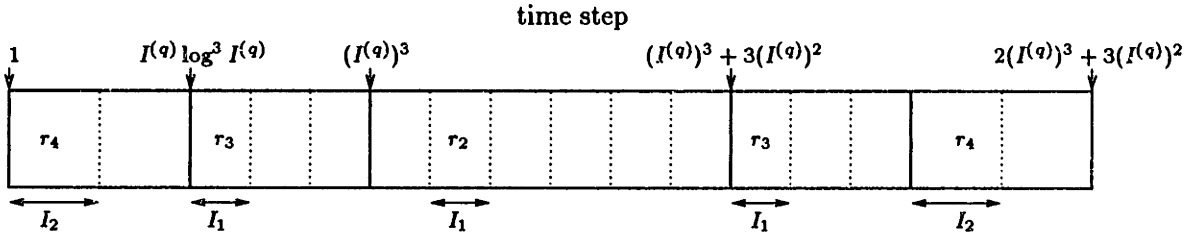


Figure 9-3: Final bounds on frame size and relative congestion. Here $I_1 = \log^2 I^{(q)}$, $I_2 = \log^4 I^{(q)}$, $r_2 = (1 + \gamma_2)c^{(q)}$, $r_3 = (1 + \gamma_3)c^{(q)}$ and $r_3 = (1 + \gamma_4)c^{(q)}$, where γ_2, γ_3 and γ_4 are $\frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$.

9.3 Conversion for Schedule $\mathcal{S}^{(q)}$

In the conversion steps we transform the schedule $\mathcal{S}^{(q+\frac{1}{2})}$, obtained from the frame-refinement step, into a new schedule $\mathcal{S}^{(q+1)}$. In this new schedule, all the sessions in $B^{(q+1)}$, which were fractional in $\mathcal{S}^{(q)}$, are made integral, and the relative congestion of frames of size $I^{(q+1)}$ or larger is at most $c^{(q+1)} = (1 + \delta^{(q)})^2 c^{(q)}$.

At the beginning of this step, we inductively assume that the relative congestion is at most $(1 + \delta^{(q)})c^{(q)}$ for any frame of size $I^{(q+1)}$ or larger, where $\delta^{(q)} = \frac{\alpha}{\sqrt{\log I^{(q)}}}$. If the set $B^{(q+1)}$ is empty then we skip this conversion step. Obviously, the relative congestion is at most $c^{(q+1)}$ for any frame of size $I^{(q+1)}$, and we are done.

On the other hand, if the set $B^{(q+1)}$ is not empty then for each session $i \in B^{(q+1)}$

we apply the following two processes. In the *discretization process*, we transform fractional sessions in $B^{(q+1)}$ into integral sessions such that these new integral packets do not wait for too long before leaving their sources. In the *delay-insertion process*, we insert *initial delays* to defer the times at which packets leave their sources in order to satisfy the relative-congestion requirement.

9.3.1 Discretization

We first present the discretization process, in which fractional sessions in $B^{(q+1)}$ are transformed into integral sessions. Consider a session i in $B^{(q+1)}$. When session i is fractional, a packet of size $\hat{r}_i = s_i/\ell_i$ is injected at every time step, where ℓ_i and s_i are integers defined in Equations 8.1 and 8.2 in Section 8.3. Each fractional packet marches to its destination one edge at a time with no delay. When these fractional packets are discretized into integral packets, each integral packet waits at its source until it finds an unused *initial token*. Then, it crosses one edge every time step until it reaches its destination. The number of initial tokens and their distribution have to be carefully chosen so that no packet waits at its source for too long.

We consider the two intervals shown in Figure 9-4, $U = [\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ and $V = [\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i)$. When session i is discretized, we distribute enough *initial tokens* in the interval V to accommodate all the session-i arrivals during U. Integral packets arrive at a rate r_i during U and initial tokens will appear at a rate at least as high as $\hat{r}_i$ during V. Recall from Section 8.3, that $\hat{r}_i$ is slightly larger than r_i. By choosing the interval U long enough (i.e. $\mathcal{T}$ large enough), we guarantee more initial tokens than arrivals.

We place initial tokens for session i in the interval V as follows. We first put s_i initial tokens for session i in the last slot of V, i.e. at time step $2\mathcal{T} - \mathcal{T}_i - 1$. For every ℓ_i steps before $2\mathcal{T} - \mathcal{T}_i - 1$, we also place s_i tokens in the same slot until we reach the beginning of V. (See Figure 9-4.) We show that there are enough initial tokens, and that no packet waits too long for an unused one.

Lemma 9.3.1 *For a discretized session $i \in B^{(q+1)}$, every session-i packet that is*

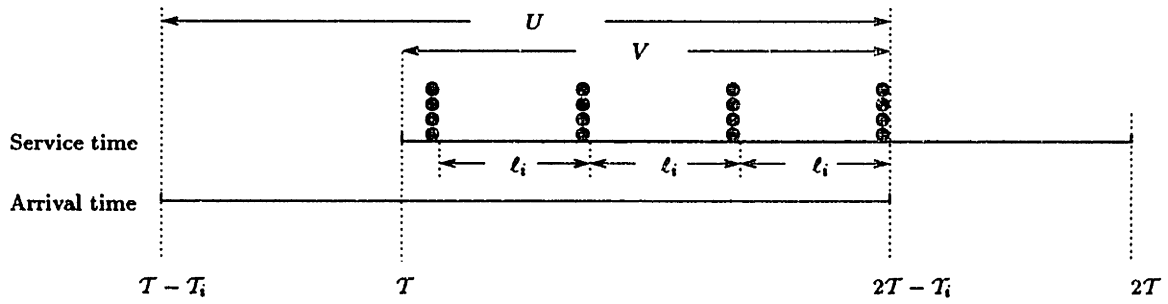


Figure 9-4: Session-i packets that are injected in interval U are assigned initial tokens in interval V. The interval V is divided into consecutive intervals of length ℓ_i, each of which has s_i initial tokens. The initial tokens are shown in solid dots.

injected during U finds an unused initial token in V within $\mathcal{T}_i + \ell_i = O(1/r_i + D_i)$ steps of its injection.

Proof: Let $x = \mathcal{T}/(\mathcal{T} - \mathcal{T}_i)$ be the ratio of the length of interval U to the length of interval V. It suffices to show that s_i, the number of initial tokens during every ℓ_i steps in V, is as large as the number of session-i arrivals during $x\ell_i$ steps. At most $n = x\ell_i r_i + 1$ packets can arrive during $x\ell_i$ steps. Since $\mathcal{T} \geq (1 + 4/\varepsilon)\max_i \mathcal{T}_i$ by definition, we have $x \leq 1 + \varepsilon/4$ and therefore $n \leq \ell_i r_i(1 + \varepsilon/4) + 1$. By the left-hand side of Inequality 8.3 in Section 8.3, we have $n \leq s_i$. Therefore, we have enough initial tokens. Since the initial tokens appear at the end of every ℓ_i steps, each packet can use an initial token that appears after the packet arrival time. It is also easy to verify that an unused initial token appears within $\mathcal{T}_i + \ell_i = O(\mathcal{T}_i)$ steps of the packet injection. □

9.3.2 Delay Insertion

Before any delay is inserted to a packet from session $i \in B^{(q+1)}$, the packet leaves its source at the time of its initial token and marches to its destination with no more waiting. Now we insert an initial delay to each session-i initial token, which has the effect of deferring the start time of session-i packets. We choose the delays randomly, uniformly and independently from $[1, \ell_i]$. After the initial delay each packet travels to its destination without further waiting.

Lemma 9.3.2 *Consider any t-frame during the interval $[\mathcal{T}, 2\mathcal{T})$ and any edge e that is used by some session $i \in B^{(q+1)}$. After the random delays are inserted, the expected number of session-i packets using edge e during the t-frame is at most $t\hat{r}_i$.*

Proof: During any time step in the interval $[\mathcal{T}, 2\mathcal{T})$, the expected number of initial tokens for session i is either 0 or $\hat{r}_i$. By linearity of expectations, the expected number of session-i tokens in any t-frame is at most $t\hat{r}_i$. Since each initial token is owned by at most one packet, the expected number of session-i packets using the first edge along the session-i path is at most $t\hat{r}_i$ for during t-frame. After obtaining the initial token, each session-i packet marches to its destination with no further waiting. Hence, the expected number of session-i packets using any subsequent edge during any t-frame is also at most $t\hat{r}_i$.

Note the above proof relies on $\mathcal{T}_i \geq \ell_i + D_i$, which is guaranteed by the definition of $\mathcal{T}_i$. More details are included in Theorem 9.4.4. □

We now use a Chernoff bound and the Lovász Local Lemma to show the following.

Lemma 9.3.3 *There exists a way of choosing the initial delays for sessions in $B^{(q+1)}$ such that the relative congestion in any frame of size $I^{(q+1)}$ or bigger is at most $c^{(q+1)}$ after the delays are inserted.*

Proof: Due to Lemma 9.2.3, it is sufficient to prove the result for all frames of size $I^{(q+1)}$ to $2I^{(q+1)}$. We associate a bad event with each edge e and each I-frame, where $I^{(q+1)} \leq I \leq 2I^{(q+1)}$. A bad event $E_{\{e,I\}}$ happens when more than $Ic^{(q+1)}$ packets use e during the I-frame. We use the Lovász Local Lemma to show that with a nonzero probability no bad event occurs.

We first bound the dependency d of bad events. The probability space is given by the delays assigned to packets from sessions in $B^{(q+1)}$. Hence, a bad event $E_{\{e,I\}}$ is dependent on another bad event $E_{\{e',I'\}}$ only if a packet from a session $i \in B^{(q+1)}$ can possibly cross edge e during the I-frame and can possibly cross e' during the I'-frame.

Let $D = \max_{i \in B^{(q+1)}} D_i$, $r = \min_{i \in B^{(q+1)}} r_i$ and $\ell = \max_{i \in B^{(q+1)}} \ell_i$. By the definition of $B^{(q+1)}$, D and $1/r$ are bounded by $e^{\sqrt{I^{(q+1)}}}$. By the definition of ℓ_i, $\ell = \Theta(1/r)$. There are at most $1/r$ sessions in $B^{(q+1)}$, each of which is at most D long. Therefore,

$E_{\{e,I\}}$ depends on $E_{\{e',I'\}}$ for at most D/r choices of e'. Furthermore, intervals I and I' cannot be more than $D+\ell$ steps apart. (Otherwise no session-i packet can possibly move during both intervals I and I', and so $E_{\{e,I\}}$ and $E_{\{e',I'\}}$ would have to be independent.) Therefore, the starting point of I' is limited to $2D+2\ell+4I^{(q+1)}$ locations, and the total possible choices for I' is at most $(2D+2\ell+4I^{(q+1)})I^{(q+1)}$. We conclude that the dependency d is $(2D+2\ell+4I^{(q+1)})I^{(q+1)}D/r$, which is $O\left(e^{4\sqrt{I^{(q+1)}}}\right)$.

We now bound the probability p that one particular bad event $\mathcal{B}_{\{e,I\}}$ happens. By our inductive assumption, the frame congestion on edge e during the I-frame is at most $(1+\delta^{(q)})c^{(q)}I$ before the conversion. Let S be the set of sessions in $B^{(q+1)}$ that use edge e. When sessions in $B^{(q+1)}$ are fractional, they contribute exactly $I\sum_{i\in S}\hat{r}_i$ to the frame congestion. Lemma 9.3.2 implies that the expected frame congestion due to the sessions in $B^{(q+1)}$ is at most $I\sum_{i\in S}\hat{r}_i$ after the initial delays are inserted. The congestion due to sessions not in $B^{(q+1)}$ does not change during the conversion. Hence, the expected frame congestion on edge e during the I-frame is at most $(1+\delta^{(q)})c^{(q)}I=\mu$. We bound the probability of $\mathcal{B}_{\{e,I\}}$ as follows.

$$\begin{aligned}
p &= \Pr\left[\text{ Frame congestion on } e \text{ in } I > c^{(q+1)}I \right] \\
&= \Pr\left[\text{ Frame congestion on } e \text{ in } I > (1+\delta^{(q)})\mu \right] \\
&\le e^{-(\delta^{(q)})^2\mu/3} \\
&\le e^{-(1-\varepsilon)\alpha^2 I^{(q+1)}/(3\log I^{(q)})} \\
&\le e^{-(1-\varepsilon)\frac{\alpha^2}{3}(I^{(q+1)})^{4/5}}
\end{aligned}$$

The first inequality follows from Lemma 7.2.1. The second inequality holds since $\mu > (1-\varepsilon)I \ge (1-\varepsilon)I^{(q+1)}$ and from the definition of $\delta^{(q)}$. The last inequality follows from the recurrence for $I^{(q+1)}$.

When α is a sufficiently large constant, we have $4dp<1$. Hence, the Lovász Local Lemma implies that with a nonzero probability no bad events occur. □

9.4 Termination at Schedule $\mathcal{S}^{(\zeta)}$

The process of refinement and conversion terminates at schedule $\mathcal{S}^{(\zeta)}$ when the frame size $I^{(\zeta)}$ becomes smaller than or equal to w, a constant defined in Section 8.3. We show that the following two lemmas hold at the termination.

Lemma 9.4.1 *In the schedule $\mathcal{S}^{(\zeta)}$ all sessions are integral.*

Proof: It is sufficient to show that the $B^{(q)}$'s form a partition of all sessions. Since $I^{(1)} = \log^2 X_{\max}$, sessions with the largest X_i values become integral during the first conversion step. Let $1 \leq q \leq \zeta - 1$. Due to the third bound in the definition of w, we have $I^{(q)} > w \geq e^{2^{2/3}}$, which implies $(I^{(q)})^2 \leq e^{\sqrt{\log^5 I^{(q)}}}$. Our lemma follows from the definition of $B^{(q)}$. □

Lemma 9.4.2 *In the schedule $\mathcal{S}^{(\zeta)}$ the relative congestion is at most $c^{(\zeta)} < 1$ for any frame of size $I^{(\zeta)}$ or larger.*

Proof: By our induction, the relative congestion is at most $c^{(\zeta)}$ for all frames of size $I^{(\zeta)}$ or larger. Hence, we only need to show that $c^{(\zeta)} < 1$. By definition,

$$c^{(\zeta)} = (1 + \delta^{(\zeta-1)})^2 (1 + \delta^{(\zeta-2)})^2 \cdots (1 + \delta^{(0)})^2 c^{(0)}.$$

Due to the first bound in the definition of w, $x < I^{(\zeta-1)}$, where x satisfies $\left(1 - \frac{\alpha}{\sqrt{\log x}}\right)^2 = 1 - \varepsilon/2$. Let $\Delta = \frac{\alpha}{\sqrt{\log x}}$, then

$$\delta^{(\zeta-1)} < \Delta < 1.$$

Due to the second bound in the definition of w, $\log^{10} I^{(q-1)} \leq I^{(q-1)}$ for $q - 1 \leq \zeta - 1$. As a result,

$$\delta^{(q-1)} = \frac{\alpha}{\sqrt{I^{(q-1)}}} < \frac{\alpha}{I^{(q)}} < (\delta^{(q)})^2.$$

Hence,

$$\begin{aligned} c^{(\zeta)} &< (1+\Delta)^2 (1+\Delta^2)^2 (1+\Delta^4)^2 (1+\Delta^8)^2 \ldots c^{(0)} \\ &\leq (1-\Delta)^{-2} \left\{ (1-\Delta)^2 (1+\Delta)^2 (1+\Delta^2)^2 (1+\Delta^4)^2 (1+\Delta^8)^2 \ldots \right\} c^{(0)} \end{aligned}$$

$$
\begin{aligned}
&\leq (1-\Delta)^{-2} c^{(0)} \\
&= \left(1 - \frac{\alpha}{\sqrt{\log x}}\right)^{-2} c^{(0)} \\
&= \frac{1-\varepsilon/2}{1-\varepsilon/2} \\
&= 1
\end{aligned}
$$

The first inequality follows from $\delta^{(q-1)} < (\delta^{(q)})^2$ and $\delta^{(\zeta-1)} < \Delta$. The third inequality holds since $\Delta < 1$, and therefore the "telescope product" in the braces is less than 1. The last equality holds by the above choice of x and the definition of $c^{(0)}$ in Section 8.3. □

Now, we have to make sure that in $\mathcal{S}^{(\zeta)}$ no packet waits for too long. The conversion step guarantees that when a session i becomes integral, no packet waits for $O(D_i + 1/r_i)$ steps before it leaves its source, and it does not wait anymore. The invariant we maintain throughout the frame-refinement steps guarantee that a moving packet never waits more than once every $I^{(\zeta-1)}$ steps by schedule $\mathcal{S}^{(\zeta)}$. However, each frame-refinement step can, in fact, delay the time an integral packet leaves its source. The following lemma shows that this delay does not add up to a large amount, and therefore that a session-i packet reaches its destination in at most $O(D_i + 1/r_i)$ steps in the schedule $\mathcal{S}^{(\zeta)}$.

Lemma 9.4.3 *During frame-refinement a session-i packet is delayed by at most $2(D_i + 1/r_i)$ steps before it leaves its source.*

Proof: Suppose session i first becomes integral in schedule $\mathcal{S}^{(q')}$. Consider a session-i packet p. For schedule $\mathcal{S}^{(q)}$, where $q \leq q' - 1$, p is never delayed during frame-refinement. For schedule $\mathcal{S}^{(q)}$, where $q \geq q'$, p is delayed by at most $I^{(q)} + (I^{(q)})^2$ steps before it starts moving. Therefore, the total delay inserted during all the frame-refinement steps is at most

$$\sum_{q \geq q'} I^{(q)} + (I^{(q)})^2.$$

Since session i becomes integral for schedule $\mathcal{S}^{(q')}$, we must $i \in B^{(q')}$. By the definition of $B^{(q')}$, $D_i + 1/r_i \geq (I^{(q')})^2$. Due to the fourth bound in the definition of w, $(I^{(q)}) +$

$(I^{(q)})^2 \geq 4(I^{(q+1)} + (I^{(q+1)})^2)$. Combining with the fact that $(I^{(q')})^2 \geq 2I^{(q')}$, we conclude,

$$\begin{aligned}\sum_{q \geq q'} I^{(q)} + (I^{(q)})^2 &\leq (D_i + 1/r_i)\left(1 + \frac{1}{2}\right)\left(1 + \frac{1}{4} + \frac{1}{4^2} + \ldots\right) \\ &\leq 2(D_i + 1/r_i).\end{aligned}$$

Hence, a session-i packet is delayed during frame-refinement by at most $2(D_i + 1/r_i)$ steps before it leaves its source. □

We proceed to prove that $\mathcal{S}^{(\zeta)}$ has the following properties.

Theorem 9.4.4 *Given network $\mathcal{M}$ and a set of sessions as defined in Section 6.1 there is a schedule $\mathcal{S}^{(\zeta)}$ such that the following hold.*

1. *The relative congestion is at most 1 for any frame of size larger than a certain constant w;*

2. *After leaving its source, each packet waits at most once every $O(1)$ steps, which implies that all edge queues in $\mathcal{M}$ have size $O(1)$;*

3. *For all sessions i, any session-i packet reaches its destination within $O(1/r_i + D_i)$ steps of its injection;*

4. *All session-i arrivals during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ are serviced during $[\mathcal{T}, 2\mathcal{T})$, i.e. all packets leave their source no earlier than $\mathcal{T}$ and reach their destination before $2\mathcal{T}$.*

Proof:

1. By Lemma 9.4.2, the relative congestion is at most 1 for any frame of size $I^{(\zeta)}$ or larger. Due to the termination condition, $I^{(\zeta)} \leq w$ is a constant.

2. By the invariant maintained throughout the frame-refinement steps, a packet waits at most once every $I^{(\zeta-1)}$ steps once it leaves its source. Furthermore, at most $I^{(\zeta)}$ packets can cross an edge simultaneously. Therefore, the edge queues have size $2I^{(\zeta)}$, which is $O(1)$.

3. We first show that a session-i packet reaches its destination within $\mathcal{T}_i$ steps after it obtains an initial token. After the initial token, a session-i packet is deferred by an initial delay during the conversion step and other delays during the frame-refinement step before it could leave its source. The initial delay is at most $\ell_i < 1 + 8/(\varepsilon r_i)$, and the delay during the refinement is at most $2(D_i + 1/r_i)$ by Lemma 9.4.3. Once the packet starts moving, it reaches its destination in at most $2D_i$ steps by Property 2. Therefore, a session-i packet reaches its destination within $4D_i + 1 + (8/\varepsilon + 2)/r_i = \mathcal{T}_i$ steps after obtaining its initial token.

   Since any session-i packet obtains an initial token within $\mathcal{T}_i + \ell_i$ steps of its injection by Lemma 9.3.1, the packet reaches its destination within $2\mathcal{T}_i + \ell_i = O(1/r_i + D_i)$ steps of its injection.

4. For all session-i arrivals during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$, the initial tokens are in $[\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i)$. From the discussion of Property 3, a session-i packet reaches its destination within $\mathcal{T}_i$ steps after it obtains an initial token. Therefore, all packets leave their sources no earlier than $\mathcal{T}$ and reach their destinations before $2\mathcal{T}$.

□

9.5 A Schedule for the Original Network

This section concentrates on Problem 3 of Chapter 8. We describe how to create a schedule $\mathcal{S}_{\mathcal{N}}$ for the original network $\mathcal{N}$ from the schedule $\mathcal{S}^{(\zeta)}$ for the intermediate network $\mathcal{M}$. Recall that in the construction of $\mathcal{M}$ from $\mathcal{N}$, each edge e in $\mathcal{N}$ is replaced by $2w$ consecutive edges $e_1, \ldots, e_{2w}$, where w is a constant defined in Section 8.3.

We first partition the time interval $[\mathcal{T}, 2\mathcal{T})$ into consecutive w-frames. (Recall that $\mathcal{T}$ is an integer multiple of w by definition.) For each w-frame and each edge f in $\mathcal{M}$, as many as w packets, $p_1, p_2, \ldots, p_w$, can cross f during the w-frame by schedule $\mathcal{S}^{(\zeta)}$. We *smooth out* $\mathcal{S}^{(\zeta)}$ so that p_j is the jth packet to cross f in the w-frame, where $p_1, \ldots, p_w$ represents an arbitrary ordering. We refer to the schedule

after the smoothing as $\mathcal{S}_{\mathcal{M}}$.

According to $\mathcal{S}_{\mathcal{M}}$, at most one packet at a time crosses each edge. However, $\mathcal{S}_{\mathcal{M}}$ may not schedule the packets to cross the edges on their routes in order. For example, $\mathcal{S}_{\mathcal{M}}$ may schedule to cross edge f before g, whereas f follows g on the route in $\mathcal{M}$. $\mathcal{S}_{\mathcal{M}}$ may also schedule a packet to leave its source before its injection time. We define $\mathcal{S}_{\mathcal{N}}$ to avoid the ordering problem. $\mathcal{S}_{\mathcal{N}}$ schedules a packet p to cross e in $\mathcal{N}$ at time t if and only if $\mathcal{S}_{\mathcal{M}}$ schedules p to cross e_{2w} in $\mathcal{M}$ at time t.

Lemma 9.5.1 *In $\mathcal{S}_{\mathcal{N}}$, each packet is scheduled to leave its source after its injection and is scheduled to cross the edges on its route in order.*

Proof: We first show that each packet crosses the edges on its route in order. Consider a packet p. Let e and $\hat{e}$ be two edges on p's route in $\mathcal{N}$, where $\hat{e}$ follows e. Let t and $\hat{t}$ be the times that p crosses e and $\hat{e}$ in schedule $\mathcal{S}_{\mathcal{N}}$. We show in the following that $t < \hat{t}$.

Let e_{2w} and $\hat{e}_{2w}$ be the edges in $\mathcal{M}$ that correspond to e and $\hat{e}$. Let τ and $\hat{\tau}$ be the times that p crosses e_{2w} and $\hat{e}_{2w}$ in the schedule $\mathcal{S}^{(\zeta)}$. Since $\mathcal{S}^{(\zeta)}$ schedules packets to cross the edges in $\mathcal{M}$ in order, we have,

$$\tau + 2w \leq \hat{\tau}. \tag{9.3}$$

In schedule $\mathcal{S}_{\mathcal{N}}$, packet p crosses e at time t, which is shifted by at most $w-1$ steps from τ. Similarly, $\hat{t}$ is shifted by at most $w-1$ steps from $\hat{\tau}$. Hence we have,

$$\begin{aligned} \tau - (w-1) &\leq t \leq \tau + (w-1), \\ \hat{\tau} - (w-1) &\leq \hat{t} \leq \hat{\tau} + (w-1). \end{aligned}$$

From Inequality 9.3 and the above inequalities, we have $t < \hat{t}$. Therefore, p crosses the edges on its route in order.

The proof that packet p leaves its source after its injection time is similar. Suppose that p is injected to the network at time s. Let edge e be the first edge on the route of p in network $\mathcal{N}$, and let t be the time that p crosses e in $\mathcal{S}_{\mathcal{N}}$. Also let e_{2w} be the

corresponding edge in $\mathcal{M}$, and let τ be the time that p crosses e_{2w} in $\mathcal{S}^{(\zeta)}$. Since $\mathcal{S}^{(\zeta)}$ schedules p to cross the edges in order and schedules p to leave its source after its injection, we have,

$$s + 2w \leq \tau.$$

In schedule $\mathcal{S}_{\mathcal{N}}$, packet p crosses e at time t, which is shifted by at most $w-1$ steps from τ. Hence we have,

$$\tau - (w-1) \leq t \leq \tau + (w-1).$$

Therefore, $s < t$ and packet p leaves its sources in $\mathcal{N}$ after the injection time. □

We summarize the properties of $\mathcal{S}_{\mathcal{N}}$.

Theorem 9.5.2 *Schedule $\mathcal{S}_{\mathcal{N}}$ satisfies the following properties.*

1. *At most one packet at a time crosses each edge in $\mathcal{N}$;*
2. *After leaving its source, each packet waits constant number of steps to cross an edge, which implies all the edge queues in $\mathcal{N}$ have constant size;*
3. *For all sessions i, any session-i packet reaches its destination within $O(1/r_i + d_i)$ steps of its injection;*
4. *All session-i arrivals during $[T - T_i, 2T - T_i)$ are serviced during $[T, 2T)$, i.e. all packets leave their source no earlier than T and reach their destination before $2T$.*

Proof: The smoothing process guarantees Property 1. Properties 2 and 3 come from Properties 2 and 3 of $\mathcal{S}^{(\zeta)}$ given in Theorem 9.4.4, the construction of $\mathcal{M}$ from $\mathcal{N}$ and the fact that each packet is scheduled to reach its destination in $\mathcal{S}_{\mathcal{N}}$ at most w steps later than in $\mathcal{S}^{(\zeta)}$.

To see Property 4, recall that the interval $[T, 2T)$ is partitioned into w-frames, and schedule $\mathcal{S}^{(\zeta)}$ is smoothed out within each w-frame. Therefore, if a packet is

scheduled to cross an edge e during $[\mathcal{T}, 2\mathcal{T})$ according to $\mathcal{S}^{(\zeta)}$, the packet must also be scheduled to cross e during $[\mathcal{T}, 2\mathcal{T})$ according to $\mathcal{S}_{\mathcal{N}}$. Property 4 implies that all intervals of $[0, \mathcal{T})$, $[\mathcal{T}, 2\mathcal{T})$, etc. can be scheduled independently. □

We now describe how to transform $\mathcal{S}_{\mathcal{N}}$ to a template-based schedule. Let $\mathcal{T}$ be the size of each template. Suppose each session-i initial token (in the conversion step of Section 9.3) is owned by a session-i packet. We have shown the existence of an optimal schedule $\mathcal{S}_{\mathcal{N}}$ for these packets using the procedure described in this chapter. The movement of each packet scheduled by $\mathcal{S}_{\mathcal{N}}$ determines a token sequence, and these token sequences define the locations of all the tokens.

Obviously, the token lag is $O(1)$ for all sequences and the end-to-end delay is $O(1/r_i + d_i)$ for all session-i token sequences. Since each session-i packet is able to obtain an initial token within $O(1/r_i + d_i)$ steps of its injection, Theorem 7.1.2 implies that the template-based schedule defined by the token sequences achieves a delay bound of $O(1/r_i + d_i)$ and constant edge queues. In summary,

Theorem 9.5.3 *Consider an arbitrary network in which sessions are defined. Each session i is associated with an injection rate r_i and path length d_i. Packets are injected to the network along these sessions subject to the injection rates. If the total rate on each edge is at most $1 - \varepsilon$ for a constant $\varepsilon \in (0, 1)$, then there exists a template-based schedule such that each session-i packet reaches its destination within $O(1/r_i + d_i)$ steps of its injection and at most one packet crosses an edge at each time step. This schedule also maintains constant edge queues.*

Chapter 10

The Construction of an Optimal Schedule

The analysis so far relies on the Lovász Local Lemma to show the *existence* of a schedule with delay bound $O(1/r_i + d_i)$. In this Chapter, we describe the key ideas to *construct* such a schedule. Since we do not present all the details, the results in this chapter are stated in terms of claims rather than theorems and lemmas.

Let us first revisit the Lovász Local Lemma. Let $E_1, \ldots, E_n$ be a set of bad events, each occurring with probability p and with dependence at most d, i.e. every bad event is mutually independent of some set of $n - d$ other bad events. If $4pd < 1$, then with a nonzero probability no bad event occurs. In other words,

$$\Pr\left[\bigcap_{i=1}^{n} \bar{E}_i\right] > 0.$$

In particular, the proof of the Local Lemma, e.g. in [54, pages 57-58], gives the lower bound,

$$\Pr\left[\bigcap_{i=1}^{n} \bar{E}_i\right] \geq \prod_{i=1}^{n}(1-2p) \approx e^{-2pn},$$

which is exponentially small. Hence, the Local Lemma does not provide an efficient randomized algorithm. The question is how to find a polynomial time algorithm for finding this exponentially small "needle" [6].

In [6], Beck converted some applications of the Local Lemma into polynomial time algorithms. In [32] Leighton et al. modified Beck's arguments to efficiently construct the optimal schedule for their static routing problem. In this chapter, our general approach is to employ the techniques in [32] to construct an optimal schedule for our dynamic problem. The construction takes time polynomial in Z and has a success probability of $1 - \frac{1}{poly(Z)}$, where

$$Z = m \max_i (1/r_i + d_i).$$

As before, m is the number of edges in the network and r_i and d_i are the injection rate and the path length of session i respectively. We also redefine the initial frame size to be,

$$I^{(0)} = me^{\log^{2/5} \max_i (1/r_i + D_i)}.$$

All other parameter definitions are the same as in Section 8.3.

Our analysis in Chapter 9 remains unchanged, except in Lemmas 9.2.4, 9.2.6 and 9.3.3 when the Lovász Local Lemma is used. We focus on the constructive version of Lemma 9.2.4, which is used for the first step of frame-refinement. Similar techniques can be applied to Lemmas 9.2.6 and Lemma 9.3.3.

10.1 Refinement

Consider refining the schedule $\mathcal{S}^{(q)}$. Recall in Section 9.2.2 we first partition the schedule into blocks of length $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}$, and reschedule each block independently. Within each block, random delays in the range of $[1..I^{(q)}]$ are inserted to each active packet. (Active packets are those integral packets in $A^{(q)}$ that advance some edge during the block.) Lemma 9.2.4 shows that a "good" set of delays can reduce the frame size from $I^{(q)}$ to $\log^2 I^{(q)}$ for the frames in the center of each block without increasing the relative congestion by much. We describe in the following how to find such delays in polynomial time with high probability.

10.1.1 High Level Ideas

To reschedule a particular block B, we define a set of bad events and a dependence graph induced by these events. For each edge g in the network and each I-frame in the block B, where $\log^2 I^{(q)} \leq I < 2\log^2 I^{(q)}$, we define a *bad event* $E_{\{g,I\}}$ that happens when the frame congestion on g during I exceeds $(1+\gamma_1)c^{(q)}I$, for some $\gamma_1 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$. These bad events form a dependence graph G, whose nodes represent the events and whose edges represent the dependencies between the events. Two events $E_{\{g,I\}}$ and $E_{\{g',I'\}}$ are dependent only if some packet can possibly cross g during I and can possibly cross g' during I'. The following claim follows from the dependence analysis in Lemma 9.2.4.

Claim 10.1.1 *Let G be the dependence graph of the bad events, then,*

1. *The number of nodes in G is at most $m(I^{(q)})^4$, which is $O(Z^2)$.*
2. *The node degree d of G is at most $(I^{(q)})^{10}$.*

Let us first describe the high-level idea. Initially, every bad event can possibly happen. The dependence graph G consists of one large connected component of $O(Z^2)$ nodes. We assign delays to the active packets one at a time until any further assignment would bring some bad event close to happening. Let P contain the active packets whose delays are assigned during this iteration, and let $\bar{P}$ contain the rest. For a bad event $E_{\{g,I\}}$ to be close to happening, the congestion on g during I must exceed its expectation by a certain amount, and therefore is unlikely. As a result of the first iteration of the delay assignment, some bad events cannot happen no matter how delays are assigned to the packets in $\bar{P}$.

Consider the subgraph of G that consists of the bad events that can still happen due to the future delay assignment to $\bar{P}$. We argue that the connected components of this subgraph have a small size of $O\left(poly(d)\log Z\right)$ with high probability. These components partition the packets in $\bar{P}$ into mutually disjoint sets. The delay assignment for each component can therefore be completed independently. There are two cases to consider. If $I^{(q)} = poly(\log Z)$, we assign random delays to all the packets in

$\bar{P}$. The combined congestion due to P and $\bar{P}$ causes no bad event to happen with high probability. If $I^{(q)} = poly(\log\log Z)$ or smaller, we carry out another iteration of delay assignment to some packets in $\bar{P}$. This further reduces the component size to $O\left(poly(d)\log\log Z\right)$. The Lovász Local Lemma shows the existence of a delay assignment to the remaining packets that causes no bad event to happen. These delays can be found through an exhaustive search in polynomial time, since the component size is small.

10.1.2 One Iteration of Delay Assignment

Let us fill in the details of the above outline. We assign delays to active packets one at a time. Each delay is chosen uniformly and independently at random from $[1, I^{(q)}]$. Suppose c active packets that can possibly use edge g during an I-frame are assigned delays so far, then a *critical event* $C_{\{g,I\}}$ happens if the frame congestion due to these c packets exceeds,

$$c\frac{I}{I^{(q)}} + \frac{k}{\sqrt{\log I^{(q)}}}\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I, \tag{10.1}$$

for an appropriately chosen constant k. Let us justify our definition of $C_{\{g,I\}}$. The first term $c\frac{I}{I^{(q)}}$ upper bounds the expected congestion on g during I due to these c packets, since each packet has probability at most $\frac{I}{I^{(q)}}$ to use g during I. The second term of (10.1) represents the extra congestion that a critical event tolerates. In particular, $\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I$ upper bounds the expected congestion due to all the packets. To see why, let r_f be the relative congestion on g due to fractional packets according to schedule $\mathcal{S}^{(q)}$, then the relative congestion due to the active packets is at most $c^{(q)} - r_f$. Hence, at most $(c^{(q)} - r_f)(I + I^{(q)})$ active packets can possibly use g during I, each of which has probability at most $\frac{I}{I^{(q)}}$ to do so. The total expected congestion is therefore $r_f I + (c^{(q)} - r_f)(I + I^{(q)})\frac{I}{I^{(q)}}$, which is at most $\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I$. In summary, when a critical event happens, the congestion due to c packets exceeds its expectation by a factor $\frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$ of the expectation due to all the packets. A Chernoff-type of argument implies,

Claim 10.1.2 *A critical event happens with probability* $(I^{(q)})^{-k_0}$, *for a positive constant* k_0 *determined by the constant* k *in (10.1).*

Proof sketch: By Lemma 7.2.1, the probability that a critical event happens is at most $e^{-(k^2/\log I^{(q)})(1+I/I^{(q)})c^{(q)}I/3}$, which is at most $e^{-(1-\varepsilon)k^2 \log I^{(q)}/3}$. □

After assigning each delay, we check to see if any critical event happens. If a packet causes a critical event $C_{\{g,I\}}$ to happen, then we set aside all other packets that can also use g during I but whose delays have not yet been assigned. Checking whether or not each critical event happens takes polynomial time in Z. The set P contains all the active packets whose delays are assigned and $\bar{P}$ contains the rest of the active packets. We now collapse the dependence graph G into much smaller connected components.

A node of G is *critical* if its corresponding event is critical, and a node is *endangered* if it is dependent of a critical event. (This implies an endangered node is always adjacent to some critical node.) Let G_1 be the subgraph of G consisting of the critical and endangered nodes and the edges connecting them. If a node is not in G_1, then all of the packets that can possibly use the corresponding edge have already been assigned delays. Therefore, the bad event corresponding to the node cannot happen no matter how we assign delays to the packets in $\bar{P}$. From now on, we only need to consider the nodes in G_1. Since any two connected components of G_1 do not share a packet, there is a one-to-one correspondence between components of G_1 and disjoint sets in a partition of the packets in $\bar{P}$. Hence, we can delay the packets in each component independently.

Initially, the dependence graph is one single large component. After the first iteration of the delay assignment, the graph collapses into much smaller pieces.

Claim 10.1.3 *The largest connected component of* G_1 *has at most* $\log X \cdot d^4$ *nodes with probability* $1 - \frac{1}{poly(X)}$, *where* $X = O(Z^2)$ *is the number of nodes in* G *and* d *is the node degree.*

The proof of this claim relies on some properties of the *cube of a graph.* For a graph H, its cube H^3 has the same nodes as H. An edge connects two nodes in H^3 if and

only if a path of at most 3 edges connects these two nodes in H. An *independent set* is a subset of the nodes in H in which no two nodes are neighbors. An independent set is *maximal* if any addition to the set makes the set no longer independent.

Observation *Let H be any graph and H^3 be its cube.*

1. *If H has node degree d, then H^3 has degree at most d^3.*
2. *If H is connected, then any maximal independent set of H is connected in H^3. Furthermore, any set that contains a maximal independent set of H is connected in H^3.*

We now sketch the proof of Claim 10.1.3.

Proof sketch: We create a sequence of graphs, G_1, G_1^3, G_2, G_2^3 and G_3, where G_1^3 and G_2^3 are cubes of G_1 and G_2 respectively, and G_2 and G_3 are subgraphs of G_1^3 and G_2^3 respectively. Our goal is to bound the largest connected component of G_3 by $\log X$ with probability $1 - \frac{1}{poly(X)}$. We then argue this is sufficient to bound the largest connected component of G_1 by $\log X \cdot d^4$ with the same probability.

The critical nodes in a connected component of G_1 are connected in G_1^3 by the Observation. This is because each endangered node of G_1 is adjacent to some critical nodes, and so the critical nodes contain a maximal independent set of G_1. Let G_2 be the subgraph of G_1^3 that consists of the critical nodes only and the edges connecting them. Note that two nodes are in the same connected component of G_2 only if they are in G_1. Let G_3 be a subgraph of G_2^3 whose nodes are induced from any maximal independent set of nodes in G_2. By the Observation, the nodes that are in the same connected component of G_2^3 are mutually independent in G_2, and hence mutually independent in G_1.

In order to bound the size of the largest connected component in G_3, we associate each component with a spanning tree. Note that two distinct connected component of G_3 have disjoint spanning trees. Let us first enumerate the different trees of t nodes in G_3. There are at most X possible roots for each tree. In a depth-first traversal of a tree starting at the root, there are d^9 ways to choose each subsequent node. This is

because the node degree of G_3 is at most d^9 by the Observation. Since each edge is traversed once in each direction and there are $t-1$ edges, the total number of trees with any one root is at most $(d^9)^{2(t-1)}$. Hence, the total number of trees of t nodes is at most Xd^{18}.

Any tree of size t in G_3 corresponds to an independent set of t critical nodes in G_1. Since each critical event happens with probability at most $(I^{(q)})^{-k_0}$ by Claim 10.1.2, the probability that all of the nodes in the independent set are critical is at most $(I^{(q)})^{-k_0 t}$. By a union bound, the probability that there exists some tree of t nodes in G_3 is at most $Xd^{18}(I^{(q)})^{-k_0 t}$. Since $d = (I^{(q)})^{10}$, the probability is $X(I^{(q)})^{-\Theta(t)}$ for a sufficiently large k_0. Note that a large k_0 can be obtained from a large k in the definition of critical events. For $t = \log X$, this probability becomes $\frac{1}{poly(X)}$.

We finish the proof by bounding the size of the largest connected component in G_1. Suppose the largest connected component of G_3 has t nodes, then the largest connected component of G_2 has at most $t \cdot d^3$ nodes. Since two critical nodes are in the same connected component of G_2 if and only if they are in G_1, each connected component of G_1 contains at most $t \cdot d^3$ critical nodes. Each critical node in G_1 has at most d endangered neighbors. Therefore, the largest connected component of G_1 is bounded by $t \cdot d^4$. Our claim follows. □

10.1.3 Schedule $\mathcal{S}^{(1)}$

We proceed to assign delays to the packets in $\bar{P}$. There are two cases to consider. If we are currently refining the schedule $\mathcal{S}^{(1)}$, the frame size is $I^{(1)} = poly(\log Z)$. Claim 10.1.3 implies that with probability $1 - \frac{1}{poly(Z)}$, the maximum component size of G_1 is $(I^{(1)})^a$, where a is a constant independent of k in (10.1) and k_0 in Claim 10.1.2. Since any two components do not share any packets from $\bar{P}$, these components correspond one-to-one to the disjoint sets in a partition of the packets in $\bar{P}$. Therefore, the delay assignment to the remaining packets are independent from component to component.

Consider a particular component U of G_1 that corresponds to a set of packets $Q \subseteq \bar{P}$. We assign a random delay chosen uniformly and independently from $[1, I^{(1)}]$

to each packet in Q. Consider a node u in U that is associated with edge g and frame I. Suppose c active packets for u are assigned during the first iteration, then at most $\bar{c}$ active packets can be assigned delays this time, where $\bar{c}$ is at most $(c^{(1)}-r_f)(I+I^{(1)})-c$ and r_f is the relative congestion on g due to the fractional packets. Claim 10.1.2 implies that with probability $(I^{(1)})^{-k_0}$, the congestion associated with node u (i.e. the congestion on g during I) due to these $\bar{c}$ packets exceeds,

$$\bar{c}\frac{I}{I^{(1)}} + \frac{k}{\sqrt{\log I^{(1)}}}\left(1+\frac{I}{I^{(1)}}\right)c^{(1)}I. \tag{10.2}$$

Since at most $(I^{(1)})^a$ nodes can be in the component U, with probability $1-(I^{(1)})^{a-k_0}$ every node in U has an associated congestion at most (10.2) due to the packets in $\bar{P}$. For a sufficiently large k_0, this probability is $1-\frac{1}{poly(\log Z)}$. If the delay assignment to Q is repeated $O\left(\frac{\log Z}{\log\log Z}\right)$ times, then the success probability for U can be enhanced to $1-\frac{1}{Xpoly(Z)}$, where $X = O(Z^2)$ is the number of nodes in G_1. We carry out the above delay assignment process for each mutually independent component of G_1. Since G_1 has at most X components, with probability $1-\frac{1}{poly(Z)}$ every node in G_1 has associated congestion at most (10.2) due to the packets in $\bar{P}$.

We now have assigned delays to all the active packets. The congestion due to the packets in P and $\bar{P}$ is (10.1) and (10.2) respectively, with probability $1-\frac{1}{poly(Z)}$. Hence, the total congestion due to all packets is bounded by,

$$\begin{aligned} & r_f I + (c+\bar{c})\frac{I}{I^{(1)}} + \frac{2k}{\sqrt{\log I^{(1)}}}\left(1+\frac{I}{I^{(1)}}\right)c^{(1)}I \\ \leq & \left(1+\frac{2k}{\sqrt{\log I^{(1)}}}\right)\left(1+\frac{I}{I^{(1)}}\right)c^{(1)}I \\ = & \left(1+\frac{\Theta(1)}{\sqrt{\log I^{(1)}}}\right)c^{(1)}I. \end{aligned}$$

In other words, no bad event happens with probability $1-\frac{1}{poly(Z)}$.

10.1.4 Schedule $\mathcal{S}^{(2)}$, etc.

If we are currently refining the schedule $\mathcal{S}^{(q)}$, for $q \geq 2$, then the frame size is $I^{(q)} = poly(\log\log Z)$ or smaller. We go through a second iteration of delay assignment in a manner similar to the first iteration. We assign random delays to the packets in $\bar{P}$ one at a time. Suppose c' packets from $\bar{P}$ that can possibly use g during I are assigned delays so far, then a critical event $C_{\{g,I\}}$ happens if the congestion due to these c' packets exceeds,

$$c'\frac{I}{I^{(q)}} + \frac{k}{\sqrt{\log I^{(q)}}}\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I \tag{10.3}$$

After each delay assignment, we check if this assignment causes any critical event to happen. If so, we set aside all other packets in $\bar{P}$ that can also use g during I but whose delays have not been assigned. Let P and P' consist of the packets whose delays are assigned delays during the first and second iteration respectively. Let $\bar{P}$ consist of the remaining active packets. Suppose G_1' is the subgraph of G_1 that consists of critical nodes and endangered nodes (due to the second iteration only) and the edges connecting them. By applying Claim 10.1.3, the largest connected component of G_1' has size at most $O\left(\log(\log Z \cdot d^4)\cdot d^4\right)$ with probability $1-\frac{1}{poly(\log Z)}$. If we repeat the second iteration $O\left(\frac{\log Z}{\log\log Z}\right)$ times, then this probability can be enhanced to $1-\frac{1}{poly(Z)}$.

Now suppose each component has $poly(\log\log Z)$ nodes. We first show the existence of a "good" delay assignment to $\bar{P}$. Consider a node u in G_1' that is associated with edge g and frame I. Suppose c packets for u are assigned delays during the first iteration and c' packets during the second iteration, then at most $\bar{c}$ packets for u remain in $\bar{P}$, where $\bar{c}$ is at most $(c^{(q)}-r_f)(I+I^{(q)})-c-c'$ and r_f is the relative congestion on g due to the fractional packets. Claim 10.1.2 implies that with probability $(I^{(q)})^{-k_0}$, the congestion on g during I due to these $\bar{c}$ packets exceeds,

$$\bar{c}\frac{I}{I^{(q)}} + \frac{k}{\sqrt{\log I^{(q)}}}\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I. \tag{10.4}$$

The dependence of G_1' is at most $(I^{(q)})^{10}$. For $k_0 = 11$, the Lovász Local Lemmas

implies the existence of some delay assignment to $\bar{P}$ such that the congestion due to $\bar{P}$ on each node in G_1' is at most (10.4).

We find the "good" delay assignment for each component in G_1' using an exhaustive search. Each packet has $I^{(q)}$ choices for delays, each node in G_1' has at most $(I^{(q)}+I)c^{(q)}$ packets that can possibly use the corresponding edge, and each component has at most $poly(\log\log Z)$ nodes. Hence, the total number of choices is $I^{(q)^{(I^{(q)}+I)c^{(q)}poly(\log\log Z)}}$ for each component. Since $(I^{(q)}+I)c^{(q)} \leq 2I^{(q)}$ and $I^{(q)} = poly(\log\log Z)$ or smaller, we need to try out at most $poly(\log\log Z)^{poly(\log\log Z)}$ delay assignments, which is $O(Z)$, for each component.

Since the first and second iterations of the delay assignment are successful with high probability, summing up (10.1), (10.3), (10.4) and $r_f I$ gives the total congestion due to all packets with high probability.

$$\begin{aligned} & r_f I + (c+c'+\bar{c})\frac{I}{I^{(q)}} + \frac{3k}{\sqrt{\log I^{(q)}}}\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I \\ \leq & \left(1+\frac{3k}{\sqrt{\log I^{(q)}}}\right)\left(1+\frac{I}{I^{(q)}}\right)c^{(q)}I \\ = & \left(1+\frac{\Theta(1)}{\sqrt{\log I^{(q)}}}\right)c^{(q)}I. \end{aligned}$$

In other words, no bad event happens with probability $1-\frac{1}{poly(Z)}$.

We have thus rescheduled one particular block in polynomial time in Z and with high probability in Z. Since there are $O(Z)$ blocks, by a union bound all the blocks can be successfully rescheduled in polynomial time with high probability. The constructive version of Lemma 9.2.4 is as follows.

Claim 10.1.4 *There exists a way of choosing delays so that in between the first and last $(I^{(q)})^2$ steps of the block B, the relative congestion of any frame of size $\log^2 I^{(q)}$ or larger is at most $(1+\gamma_1)c^{(q)}$, for some $\gamma_1 = \frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$. This set of delays can be found in time polynomial in Z with probability $1-\frac{1}{poly(Z)}$.*

Similar statement can be made for the constructive version of Lemma 9.2.6.

10.2 Conversion

We now consider the conversion step of the schedule $\mathcal{S}^{(q)}$. Recall in Section 9.3 we inductively assume the frame size is $I^{(q+1)}$ and the relative congestion is $(1+\delta^{(q)})c^{(q)}$. Our goal is to convert the sessions in $B^{(q+1)}$ from fractional to integral while maintaining the frame size of $I^{(q+1)}$ and bounding the relative congestion by $c^{(q+1)} = (1+\delta^{(q)})^2 c^{(q)}$. We achieve this by first discretizing the sessions in $B^{(q+1)}$ and then inserting initial delays to these newly-converted packets. The sessions not in $B^{(q+1)}$ remain unaffected during the conversion step. Lemma 9.3.3 shows the existence of a "good" set of delays. In this section, we aim to find these delays efficiently.

We first define a set of bad events $E_{\{g,I\}}$ for each edge g and each I-frame in the interval $[T, 2T)$, where $I^{(q+1)} \leq I < 2I^{(q+1)}$. the bad event $E_{\{g,I\}}$ happens when the total congestion due to all packets exceeds $c^{(q+1)}I$, where $c^{(q+1)} = (1+\delta^{(q)})^2 c^{(q)}$.

Schedule $\mathcal{S}^{(0)}$

For the initial schedule $\mathcal{S}^{(0)}$, the argument is simple. By the analysis of Lemma 9.3.3 the probability that one particular bad event on edge g and frame I happens is $e^{-\Theta(\log^4 I^{(0)})}$. Since the total number of bad events is at most $mTI^{(1)} = O(Z^2)$, by a union bound no bad event happens with probability $1 - \frac{1}{poly(Z)}$. This explains why the definition of $I^{(0)}$ in this chapter is different from the definition in Section 8.3.

One Iteration of Delay Assignment

For the conversion step for the schedule $\mathcal{S}^{(q)}$, where $q \geq 1$, we adapt the techniques used for the refinement step in Section 10.1. The analysis is almost identical, except for the size and the node degree of the dependence graph, the definition of a critical event and the probability that a critical event happens. The following claim follows from the dependence analysis in Lemma 9.3.3.

Claim 10.2.1 *Let G be the dependence graph of the bad events, then,*

1. *The number of nodes in G is $mTI^{(q+1)}$, which is $O(Z^2)$.*

2. *The node degree d of G is $O\left(e^{4\log^{5/2} I^{(q)}}\right)$.*

We assign initial delays, chosen uniformly and independently from $[1..\ell_i]$, to as many packets in $B^{(q+1)}$ as possible until critical events happen. (Recall $\ell_i = \Theta(1/r_i)$ is defined in Section 8.3.) Suppose c packets from $B^{(q+1)}$ that can possibly use edge g during an I-frame are assigned delays so far. A critical event $C_{\{g,I\}}$ happens if the congestion due to these c packets exceeds,

$$\mu_c + \frac{k}{\sqrt{\log I^{(q)}}}(1+\delta^{(q)})c^{(q)}I.$$

The first term μ_c is the expected congestion due to these c packets. Due to the analysis in Lemma 9.3.3, the quantity $(1+\delta^{(q)})c^{(q)}I$ upper bounds the expected congestion due to all packets. (This same quantity also upper bounds the congestion on e during I at the beginning of this conversion step.) Hence, a critical event $C_{\{g,I\}}$ happens when the congestion due to c packets exceeds its expectation by a factor $\frac{\Theta(1)}{\sqrt{\log I^{(q)}}}$ of the expectation due to all the packets. A Chernoff-type of argument implies,

Claim 10.2.2 *A critical event happens with probability $e^{-\Theta(\log^4 I^{(q)})}$.*

Let G_1 be a subgraph of G that contains the critical and endangered nodes and the edges connecting them. The following result follows from a proof analogous to Claim 10.1.3.

Claim 10.2.3 *The largest connected component of G_1 has size at most $\log X \cdot d^4$ with probability $1 - \frac{1}{poly(X)}$, where $X = O(Z^2)$ is the number of nodes in G and d is the node degree.*

Let P contain all the packets in $B^{(q+1)}$ whose delays are assigned during the first iteration, and let $\bar{P}$ contain the remaining packets in $B^{(q+1)}$. Since any two components of G_1 do not share any packets from $\bar{P}$, these components correspond one-to-one to the disjoint sets in a partition of the packets in $\bar{P}$. Therefore, the delay assignment to the packets in $\bar{P}$ are independent from component to component.

Schedule $\mathcal{S}^{(1)}$

For schedule $\mathcal{S}^{(1)}$, we assign random delays from $[1..\ell_i]$ to all the packets in $\bar{P}$. Suppose for edge g and frame I, the expected congestion due to $\bar{P}$ is $\mu_{\bar{c}}$. By Claim 10.2.2, the congestion due to $\bar{P}$ exceeds,

$$\mu_{\bar{c}} + \frac{k}{\sqrt{\log I^{(1)}}}(1+\delta^{(1)})c^{(1)}I, \tag{10.5}$$

with probability $e^{-\Theta(\log^4 I^{(1)})}$. Claim 10.2.3 implies that the largest connected component of G_1 contains at most $poly(I^{(1)})e^{\Theta(\log^{5/2} I^{(1)})}$ nodes. For each particular component U of G_1, the congestion due to $\bar{P}$ on each node in U is at most (10.5) with probability $1 - e^{-\Theta(\log^4 I^{(1)})}poly(I^{(1)})e^{\Theta(\log^{5/2} I^{(1)})}$, which is at most $1 - \frac{1}{poly(I^{(1)})}$. This probability can be enhanced to $1 - \frac{1}{poly(Z)}$ by repeating the assignment $O\left(\frac{\log Z}{\log\log Z}\right)$ times.

Since the conversion step does not affect the sessions not in $B^{(2)}$, the sum of μ_c, $\mu_{\bar{c}}$ and the congestion due to sessions not in $B^{(2)}$ is at most $(1+\delta^{(1)})c^{(1)}I$. Hence, the total congestion on any node of the dependence graph G is bounded by $\left(1 + \frac{2k}{\sqrt{\log I^{(1)}}}\right)(1+\delta^{(1)})c^{(1)}I$ with probability $1 - \frac{1}{poly(Z)}$. In other words, no bad event happens with probability $1 - \frac{1}{poly(Z)}$.

Schedule $\mathcal{S}^{(2)}$, etc.

For schedules $\mathcal{S}^{(q)}$, where $q \geq 2$, we go through another iteration of delay assignment to the packets in $\bar{P}$ until critical events due to $\bar{P}$ happens. By applying Claim 10.2.3, we can now argue that the largest component size is $\log(\log Z \cdot d^4) \cdot d^4$, which is $O(\log\log Z e^{\Theta(\log^{5/2} I^{(q)})})$ with probability $1 - \frac{1}{poly(\log Z)}$. By repeating this second iteration $O\left(\frac{\log Z}{\log\log Z}\right)$ times, the probability can be enhanced to $1 - \frac{1}{poly(Z)}$.

We first show the existence of a "good" delay assignment to the remaining packets in $\bar{P}$. Suppose a node is associated with an edge g and a frame I, and $\bar{c}$ remaining packets can still use g during I. By Claim 10.2.2, the congestion due to these $\bar{c}$ packets exceeds $\mu_{\bar{c}} + \frac{k}{\sqrt{\log I^{(q)}}}(1+\delta^{(q)})c^{(q)}$ with probability at most $e^{-\Theta(\log^4 I^{(q)})}$. Since

the dependence is $O\left(e^{4\log^{5/2} I^{(q)}}\right)$ by Claim 10.2.1, the Lovász Local Lemma implies the existence of a delay assignment for the remaining packets in $\bar{P}$ such that they incur a congestion of at most $\mu_{\bar{c}} + \frac{k}{\sqrt{\log I^{(q)}}}(1+\delta^{(q)})c^{(q)}$ on each node. Note that this congestion bound guarantees that the total congestion due to all packets is at most $\left(1+\frac{3k}{\sqrt{I^{(q)}}}\right)(1+\delta^{(q)})c^{(q)}$ with probability $1-\frac{1}{poly(Z)}$.

We obtain such a delay assignment through an exhaustive search. Let $\ell = \max_{i\in B^{(q+1)}} \ell_i$, then $\ell = O\left(e^{\log^{5/2} I^{(q)}}\right)$. For a packet from session $i \in B^{(q+1)}$, the total number of choices for delay is at most ℓ. Each node in a component has at most x packets from $B^{(q+1)}$ that can possibly use the corresponding edge, where $x = \sum_{i\in B^{(q+1)}} (1+\lceil I/\ell_i \rceil)\, s_i$ is $O(I+\ell)$. (Recall $s_i = \Theta(1)$ is defined in Section 8.3.) Since each component has $O(\log\log Ze^{\Theta(\log^{5/2} I^{(q)})})$ nodes, the total number of tries for one component is at most $\ell^{O(I+\ell)O(\log\log Ze^{\Theta(\log^{5/2} I^{(q)})})}$, which is $O(Z)$.

The constructive version of Lemma 9.3.3 is as follows.

Claim 10.2.4 *There exists a way of choosing the initial delays for sessions in $B^{(q+1)}$ such that the relative congestion in any frame of size $I^{(q+1)}$ or bigger is at most $c^{(q+1)}$ after the delays are inserted. This set of delays can be found in time polynomial in Z with probability $1-\frac{1}{poly(Z)}$.*

Chapter 11

Conclusions

In Part I of this thesis we presented methods for latency hiding in simple networks such as linear arrays and 2-dimensional arrays. Ultimately, we are interested in the efficient implementation of algorithms designed for networks that appear often in the architectures of parallel computers, such as trees, arrays, butterflies and hypercubes, on a network with arbitrary topology and arbitrary link delays, such as NOWs. The special case in which two networks have identical topology but different link delays is a starting point where we can study the effect of latencies in isolation. Indeed, the general case of simulating a unit-delay guest on a host with arbitrary delays and arbitrary topology so as to minimize slowdown seems a very challenging problem.

In Part II of this thesis we presented an asymptotically-optimal schedule for a dynamic packet routing problem in connection-oriented networks. Our research demonstrates the power of randomness and synchronization among switches. Much future work can be done on this problem. For example, it would be useful to reduce the complexity of the analysis and the hidden constant in our bounds. It would also be interesting to see if our scheme can be made distributed while maintaining the optimal performance guarantees. Scheduling in a more general model that allows packets of nonuniform sizes, switches with different processing powers and traffic streams with fluctuating rates [4, 9] would be another intriguing problem.

Bibliography

[1] *The Connection Machine CM-5 Technical Summary.* Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264, 1991.

[2] F. Afrati, C. H. Papadimitriou, and G. Papageorgious. Scheduling dags to minimize time and communication. Technical report, National Technical University of Athens, Athens, Greece, 1985.

[3] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A case for NOW (networks of workstations). Technical report, University of California, Berkeley, 1994.

[4] M. Andrews, B. Awerbuch, A. Fernandez, J. Kleinberg, T. Leighton, and Z. Liu. Universal stability results for greedy contention-resolution protocols. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 380 – 389, Burlington, VT, October 1996.

[5] Y. Aumann and M. Ben-Or. Computing with faulty arrays. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 162–169, 1992.

[6] J. Beck. An algorithmic approach to the Lovász Local Lemma I. *Random Structures and Algorithms*, 2(4):343 – 365, 1991.

[7] J. Bennett and H. Zhang. Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96*, pages 120 – 128, 1996.

[8] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Fourth ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming PPoPP, San Diego, CA, pages 102–112. ACM Press, New York, NY, 1993.

[9] A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson. Adversarial queueing theory. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, Philadelphia, PA, May 1996.

[10] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493 – 509, 1952.

[11] P. Chretienne. A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European Journal of Operational Research*, 43:225 – 230, 1989.

[12] R. Cole, B. Maggs, and R. Sitaraman. Multi-scale self-simulation: A technique for reconfiguring arrays with faults. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 561–572, 1993.

[13] J. Y. Colin and P. Chretienne. C.P.M. scheduling with small communication delay and task duplication. *Technical Notes*, pages 680 – 684, 1990.

[14] R. L. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, pages 114 – 31, 1991.

[15] R. L. Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, pages 132 – 141, 1991.

[16] J. Davin and A. Heybey. A simulation study of fair queueing and policy enforcement. *Communication Review*, pages 23 – 29, 1990.

[17] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3 – 26, 1990.

[18] A. Elwalid, D. Mitra, and R. H. Wentworth. A new approach for allocating buffers and bandwidth to heterogeneous, regulated traffic in an ATM node. *IEEE Journal on selected areas in communications*, pages 1115 – 1127, 1995.

[19] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of IEEE INFOCOM'94*, pages 636 – 646, June 1994.

[20] S. J. Golestani. Congestion-free transmission of real-time traffic in packet networks. In *Proceedings of IEEE INFOCOM'90*, pages 527 – 536, 1990.

[21] S. J. Golestani. A framing strategy for connection management. In *Proceedings of IEEE SIGCOM'90*, 1990.

[22] S. J. Golestani. Duration-limited statistical multiplexing of delay sensitive traffic in packet networks. In *Proceedings of IEEE INFOCOM'91*, 1991.

[23] E. Hahne. *Round robin scheduling for fair flow control.* PhD thesis, MIT, 1986.

[24] D. N. Jayasimha and M. C. Loui. The communication complexity of parallel algorithms. Technical report CSRD 629, University of Illinois at Urbana-Champaign, 1986.

[25] H. Jurgen, L. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling for dags with communications delays. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 254–264, Santa Fe, NM, 1989.

[26] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 285–296, 1990.

[27] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOM'91*, pages 3 – 15, September 1991.

[28] S. Keshav. *An engineering approach to computer networking.* Addison Wesley, 1997.

[29] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications.* Wiley, New York, 1976.

[30] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 227–240, 1989.

[31] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Combinatorica*, 14(2):167 – 186, 1994.

[32] F. T. Leighton, B. M. Maggs, and A. W. Richa. Fast algorithms for finding O(congestion + dilation) packet routing schedules. Technical report CMU-CS-96-152, Carnegie Mellon University, 1996.

[33] F. T. Leighton and G. Plaxton. Hypercubic sorting networks. *SIAM Journal of Computing (to appear)*, 1997.

[34] T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[35] T. Leighton, B. Maggs, and R. Sitaraman. On the fault tolerance of some popular bounded-degree networks. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 542–552, 1992.

[36] C. E. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The network architecture of the connection machine CM-5. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[37] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 704–713, 1993.

[38] C. Lu and P. R. Kumar. Distributed scheduling based on due dates and buffer prioritization. Technical report, University of Illinois, 1990.

[39] D. Mitra and I. Ziedins. Virtual partitioning by dynamic priorities: Fair and efficient source-sharing by several services. In *Proceedings of International Zurich Seminar in Digital Communications*, 1996.

[40] J. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, pages 435 – 438, 1987.

[41] R. Ostrovsky and Y. Rabani. Local control packet switching algorithm. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (to appear)*, May 1997.

[42] M. Palis, J-C Liou, S. Rajasekaran, S. Shende, and D. L. Wei. On-line scheduling of dynamic trees. *Manuscript*, 1994.

[43] M. Palis, J-C Liou, and D. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. Technical report Fukushima 965-80, University of Aizu, Japan, 1994.

[44] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM Journal of Computing*, 16(4):639 – 646, 1987.

[45] C. H. Papadimitriou and M. Yannakakis. Toward an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322 – 328, 1990.

[46] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344 – 357, 1993.

[47] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple-node case. *IEEE/ACM Transactions on Networking*, 2(2):137 - 150, 1994.

[48] J. R. Perkins and P. R. Kumar. Stable distributed real-time scheduling of flexible manufacturing systems. *IEEE Transactions on Automatic Control*, pages 139 - 148, 1989.

[49] Y. Rabani and E. Tardos. Distributed packet switching in arbitrary networks. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, Philadelphia, PA, May 1996.

[50] M. O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[51] S. Shenker. Making greed work in networks: A game theoretical analysis of switch service disciplines. In *Proceedings of ACM SIGCOM'94*, pages 47 - 57, August 1994.

[52] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of ACM SIGCOM'95*, September 1995.

[53] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE*, pages 298:241–248, 1981.

[54] J. Spencer. *Ten Lectures on the Probabilistic Methods.* Capital City Press, Philadelphia, Pennsylvania, 1994.

[55] D. Stailadis and A. Varma. Frame-based fair queueing: A new traffic scheduling algorithm for packet-switched networks. Technical report UCSD-CRL-95-39, University of California at Santa Cruz, July 1995.

[56] D. Stailadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. In *Proceedings of the conference on Computer Communications, INFOCOM'96*, pages 111 - 119, March 1996.

[57] Lewis W. Tucker and George G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–38, 1988.

[58] J. S. Turner. New directions in communications, or which way to the information age. *IEEE Communications Magazine*, pages 8 – 15, 1986.

[59] L. G. Valiant. Bulk-synchronous parallel computers. Technical report TR-08-89, Center for Research in Computing Technology, Harvard University, 1989.

[60] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[61] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, pages 101 – 124, May 1991.