10-1985

Randomized Routing on Fat-trees

Ronald I. Greenberg
Rgreen@luc.edu

Author Manuscript

This is a pre-publication author manuscript of the final, published article.

Recommended Citation

Greenberg, Ronald I.. Randomized Routing on Fat-trees. Proceedings of the 26th Annual Symposium on Foundations of Computer Science, , : 241-249, 1985. Retrieved from Loyola eCommons, Computer Science: Faculty Publications and Other Works, http://dx.doi.org/10.1109/SFCS.1985.46

Randomized Routing on Fat-Trees

(Preliminary Version)

Ronald I. Greenberg
Charles E. Leiserson

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Fat-trees are a class of routing networks for hardware-efficient parallel computation. This paper presents a randomized algorithm for routing messages on a fat-tree. The quality of the algorithm is measured in terms of the *load factor* of a set of messages to be routed, which is a lower bound on the time required to deliver the messages. We show that if a set of messages has load factor $\lambda = \Omega(\lg n \lg\lg n)$ on a fat-tree with n processors, the number of delivery cycles (routing attempts) that the algorithm requires is $O(\lambda)$ with probability $1 - O(1/n)$. The best previous bound was $O(\lambda \lg n)$ for the off-line problem where switch settings can be determined in advance. In a VLSI-like model where hardware cost is equated with physical volume, we use the routing algorithm to demonstrate that fat-trees are universal routing networks in the sense that any routing network can be efficiently simulated by a fat-tree of comparable hardware cost.

1 Introduction

Fat-trees constitute a class of routing networks for general-purpose parallel computation. This paper presents a randomized algorithm for routing a set of messages on a fat-tree. The routing algorithm and its analysis generalize an earlier *universality* result by showing, in a three-dimensional VLSI model, that for a given volume of hardware, a fat-tree is nearly the best routing network that can be built. This universality result had been proved only for *off-line* simulations [8], where switch settings can be determined in advance; this paper extends it to the more interesting *on-line* case, where messages are spontaneously generated by processors.

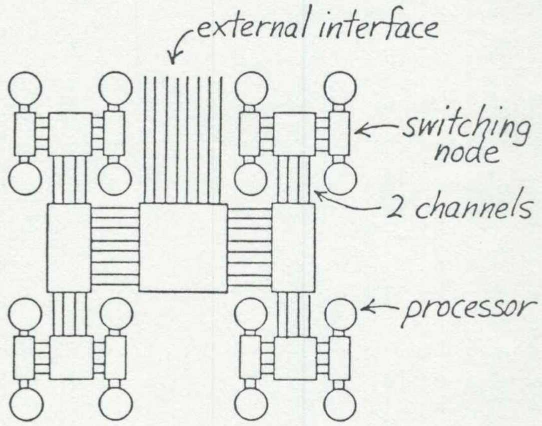

Figure 1: The organization of a fat-tree. Processors are located at the leaves, and the internal nodes contain concentrator switches. The capacities of channels increase as we go up the tree.

As is illustrated in Figure 1, a fat-tree is a routing network based on Leighton's tree-of-meshes graph [7]. A set of n processors are located at the leaves of a complete binary tree. Each edge of the underlying tree corresponds to two *channels* of the fat-tree: one from parent to child, the other from child to parent. Unlike a normal tree interconnection which is "skinny all over," each channel of a fat-tree consists of a bundle of wires. The number of wires in a channel c is called its capacity, denoted by $\text{cap}(c)$. Each internal node of the fat-tree contains circuitry that switches messages from incoming to outgoing channels.

The capacities of the channels in a fat-tree determine how much hardware is required to build it, where we measure hardware in terms of three-dimensional volume. The greater the capacities of the channels, the greater the communication potential, and also, the greater the volume. The capacities in a *universal* fat-tree [8] grow exponentially as we go from leaves to root, where the

This research was supported in part by the Defense Advanced Research Projects Agency under Contract N00014-80-C-0622. Ron Greenberg is supported in part by a Fannie and John Hertz Foundation Fellowship. Charles Leiserson is supported in part by an NSF Presidential Young Investigator Award.

base of the exponential is at most 2. Section 5 shows that for a given amount of hardware, a universal fat-tree is nearly the best network that can be built.

We shall consider communication through the fat-tree network to be synchronous, bit serial, and batched. By synchronous, we mean that the system is globally clocked. By bit serial, we mean that the messages can be thought of as bit streams. Each message snakes its way through the wires and switches of the fat-tree, with leading bits of the message setting switches and establishing a path for the remainder to follow. By batched, we mean the messages are grouped into *delivery cycles*. During a delivery cycle, the processors send messages through the network. Each message attempts to establish a path from its source to its destination. Since some messages may be unable to establish connections during a delivery cycle, each successfully delivered message is acknowledged through its communication path at the end of the cycle. Rather than buffering undelivered messages, we simply allow them to try again in a subsequent delivery cycle. The routing algorithm is responsible for grouping the messages into delivery cycles so that all the messages are delivered in as few cycles as possible.

The mechanics of routing messages in a fat-tree are almost as simple as routing in an ordinary tree. For each message, there is a unique path from its source processor to its destination processor in the underlying complete binary tree, which can be specified by a relative address consisting of at most $2 \lg n$ bits telling whether the message turns left or right at each internal node.

Within each node of the fat-tree, the messages destined for a given output channel are *concentrated* onto the available wires of that channel. This concentration may result in "lost" messages if the number of messages destined for the output channel exceeds the capacity of the channel. We assume, however, that the concentrators within the node are ideal in the sense that no messages are lost if the number of messages destined for a channel is less than or equal to the capacity of the channel. Such a concentrator can be built, for example, with a log-depth sorting network [1]. A more practical log-depth circuit can be built by combining a parallel prefix circuit [6] with a butterfly (a. k. a. FFT, Omega) network.

The performance of any routing algorithm depends on the locality of communication in a set of messages because some messages may be routed locally within subtrees of the fat-tree without soaking up bandwidth near the root. The locality of communication for a message set M can be summarized by a measure $\lambda(M)$ called the *load factor*, which we define in a more general network setting.

load factor	delivery cycles
$0 \leq \lambda(M) \leq 1$	1
$1 \leq \lambda(M) \leq 2$	$O(\lg n)$
$2 \leq \lambda(M) \leq \lg n \lg \lg n$	$O(\lg n \lg(\lambda(M)))$
$\lg n \lg \lg n \leq \lambda(M) \leq \text{poly}(n)$	$O(\lambda(M))$

Figure 2: Number of delivery cycles required to deliver a message set M on a fat-tree with n processors. All bounds are achieved with probability $1 - O(1/n)$. We assume in line 4 that the load factor $\lambda(M)$ is polynomially bounded.

Definition: Let R be a routing network. A set S of wires in R is a (directed) *cut* if it partitions the network into two sets of processors A and B such that every path from a processor in A to a processor in B contains a wire in S. The *capacity* $\text{cap}(S)$ is the number of wires in the cut. For a set of messages M, define the *load* $\text{load}(M, S)$ of M on a cut S to be the number of messages in M that must cross S. The *load factor* of M on S is

$$\lambda(M,S) = \frac{\text{load}(M,S)}{\text{cap}(S)} ,$$

and the *load factor* of M on the entire network R is

$$\lambda(M) = \max_S \lambda(M,S) .$$

The load factor provides a simple lower bound on the number of delivery cycles required to deliver a set of messages. When the set of messages is known in advance, it has been shown [8] that a set M of messages can be delivered in $O(\lambda(M) \lg n)$ delivery cycles on a fat-tree with n processors. Our routing algorithm, whose running time is summarized in Figure 2, improves this *off-line* result in two ways. First, the algorithm does not need to know the set of messages in advance, but can deliver them *on-line*. Second, the bounds on running time generally improve (and always at least match) the previous off-line bound. The only caveat is that our algorithm is randomized instead of being deterministic, but the stated bounds are achieved with high probability.

The analysis in terms of load factor is not restricted to permutation routing or situations where each processor can only send or receive a constant number of messages, as is common in the literature. We consider the general situation where each processor can send and receive polynomially many messages. Furthermore, we make no assumptions about the statistical distribution of messages, except insofar as they affect the load factor.

Our routing algorithm also differs from others in the literature in the way randomization is used. Unlike the algorithms of Valiant [11], Valiant and Brebner [12], Aleliunas [2], Upfal [10] and Pippenger [9], for example, it does not randomize with respect to paths taken by messages. Instead, for each delivery cycle, each undelivered message randomly chooses whether to be sent.

The remainder of this paper is organized as follows. Section 2 describes the randomized algorithm for routing on fat-trees. Section 3 contains some preliminary lemmas needed to analyze the algorithm, and Section 4 contains the full analysis. Section 5 contains a variety of results that follow from the randomized routing algorithm. It shows how the universality result of [8] can be extended to on-line simulations, and it includes a modification of the routing algorithm which achieves better bounds when each channel has capacity $\Omega(\lg n)$. It also gives an existential lower bound for the naive greedy approach to routing messages which shows that the greedy strategy is inferior to the randomized algorithm for worst case inputs. Finally, Section 6 contains some concluding remarks.

2 The routing algorithm

This section gives a randomized algorithm for routing a set M of messages, which is based on routing random subsets of the messages in M. The algorithm *RANDOM* is shown in Figure 3, and it uses the subroutine *TRY-GUESS* shown in Figure 4. Section 4 will provide a proof that on an n-processor fat-tree, the probability is at least $1 - O(1/n)$ that *RANDOM* delivers all messages in M within the number of delivery cylces specified by Figure 2, if the two constants k_1 and k_2 appearing in the algorithm are properly chosen.

The basic idea of *RANDOM* is to pick a random subset of messages to send in each delivery cycle by independently choosing each message with some probability p. This idea is sufficiently important to merit a formal definition.

> **Definition:** A *p-subset* of M is a subset of M formed by independently choosing each message of M with probability p.

We will show in Section 4 that if p is sufficiently small, a substantial portion of the messages in a p-subset are delivered because they encounter no congestion during routing. On the other hand, if p is too small, few messages are sent. *RANDOM* varies the probability p from cycle to cycle, seeking random subsets of M which contain a substantial portion of the messages in M but which do not cause congestion.

The algorithm *RANDOM* varies the probability p because the load factor $\lambda(M)$ is not known. The over-

```
send $M$
$U \leftarrow M - \{\text{messages delivered}\}$
$\lambda_{guess} \leftarrow 2$
while $k_1\lambda_{guess} < k_2 \lg n$ and $U \neq \emptyset$ do
    $TRY\text{-}GUESS(\lambda_{guess})$
    $\lambda_{guess} \leftarrow \lambda_{guess}^2$
endwhile
$\lambda_{guess} \leftarrow (k_2/k_1)\lg n \lg\lg n$
while $U \neq \emptyset$ do
    $TRY\text{-}GUESS(\lambda_{guess})$
    $\lambda_{guess} \leftarrow 2\lambda_{guess}$
endwhile
```

Figure 3: The randomized algorithm *RANDOM* for delivering a message set M on a fat-tree with n processors. This algorithm achieves the running times in Figure 2 with high probability if the constants k_1 and k_2 are appropriately chosen. Since the load factor $\lambda(M)$ is not known in advance, it is necessary to make guesses, each one being tried out by the subroutine *TRY-GUESS*.

```
procedure $TRY\text{-}GUESS(\lambda_{guess})$
$\lambda \leftarrow \lambda_{guess}$
while $\lambda > 1$ do
    for $i \leftarrow 1$ to $\max\{k_1\lambda, k_2 \lg n\}$ do
        independently send each message of $U$
        with probability $1/r\lambda$
        $U \leftarrow U - \{\text{messages delivered}\}$
    endfor
    $\lambda \leftarrow \lambda/2$
endwhile
send $U$
$U \leftarrow U - \{\text{messages delivered}\}$
```

Figure 4: The subroutine *TRY-GUESS* used by the algorithm *RANDOM* which tries to deliver the set U of currently undelivered messages. When $\lambda_{guess} \geq \lambda(U)$, this attempt will be successful with high probability, if the constants k_1 and k_2 are appropriately chosen. (The value r is the congestion parameter of the fat-tree defined in Section 4, which is typically a small constant.) In that case, λ is always an upper bound on $\lambda(U)$, which is at least halved in each iteration of the **while** loop. When the loop is finished, $\lambda(U) \leq 1$, so all the remaining messages can be sent.

all structure of *RANDOM* is to guess the load factor and call the subroutine *TRY-GUESS* for each one. *TRY-GUESS* determines the probability p based on *RANDOM*'s guess λ_{guess} and a parameter r, called the *congestion parameter*, which will be defined in Section 4. If λ_{guess} is an upper bound on the true load factor $\lambda(M)$, each iteration of the **while** loop in *TRY-GUESS* halves λ_{guess} with high probability, as will be shown in Section 4. When the loop is finished, we have $\lambda(U) \leq \lambda_{guess} \leq 1$, and all the remaining messages can be delivered in one cycle. The number of delivery cycles performed by *TRY-GUESS* is $O(\lg \lambda_{guess} \lg n)$ if $2 \leq \lambda_{guess} \leq \Theta(\lg n)$, and the number of cycles is $O(\lambda_{guess} + \lg n \lg \lg n)$ if $\lambda_{guess} = \Omega(\lg n)$.

RANDOM must make judicious guesses for the load factor because *TRY-GUESS* may not be effective if its guess is smaller than the true load factor. Conversely, if the guess is too large, too many delivery cycles will be performed. Since the amount of work done by *TRY-GUESS* grows as $\lg \lambda_{guess}$ for λ_{guess} small, and as λ_{guess} for λ_{guess} large, there are two main phases to *RANDOM*'s guessing. (These phases follow the handling of very small load factors, i.e., $\lambda(M) \leq 2$.)

In the first phase, the guesses are squared from one trial to the next. Once λ_{guess} is sufficiently large, we move into the second phase, and the guesses are doubled from one trial to the next. In each phase, the number of delivery cycles run by *TRY-GUESS* from one call to the next forms a geometric series. Thus, the work done in any call to *TRY-GUESS* is only a constant factor times all the work done prior to the call. With this guessing strategy, we can deliver a message set using only a constant factor more delivery cycles than would be required if we knew the load factor in advance.

3 Preliminary lemmas

This section contains three lemmas that will be needed to analyze the algorithm *RANDOM* from the preceding section. The first lemma relates the definition of load factor given in Section 1 to the channel structure of the fat-tree. The other two are technical lemmas concerning basic probability. One is a combinatorial bound on the tail of the binomial distribution of the kind attributed to Chernoff [4], and the other is a more general, but weaker, bound on the probability that a random variable takes on values smaller than the expectation.

The first lemma states that in a fat-tree, the load factor of a set of messages is determined by the cuts on the channels alone.

Lemma 1 *The load factor of a set M of messages on a fat-tree is*

$$\lambda(M) = \max_c \lambda(M,c) \; ,$$

where c ranges over all channels of the fat-tree. ■

The next lemma is a "Chernoff" bound on the tail of a binomial distribution. Suppose that we have t independent Bernoulli trials, each with probability p of success. It is well known [5] that the probability that there are at least s successes out of the t trials is

$$B(s,t,p) = \sum_{k=s}^{t} \binom{t}{k} p^k (1-p)^{t-k} \; .$$

The lemma bounds the probability that the number of successes is larger than the expectation pt.

Lemma 2

$$B(s,t,p) \leq \left(\frac{ept}{s}\right)^s \; . \blacksquare$$

The final lemma in this section bounds the probability that a bounded random variable takes on values smaller than the expectation.

Lemma 3 *Let $X \leq b$ be a random variable with expectation μ. Then for any $w < \mu$, we have*

$$\Pr\{X \leq w\} \leq 1 - \frac{\mu - w}{b - w} \; . \blacksquare$$

4 Analysis of the routing algorithm *RANDOM*

This section contains the analysis of *RANDOM*, the routing algorithm for fat-trees presented in Section 2. We shall show that the probability is $1 - O(1/n)$ that *RANDOM* delivers a set of M of messages on a universal fat-tree with n processors in the number of delivery cycles given by Figure 2

We begin by analyzing the routing of a p-subset M' of a set M of messages. If the number $\text{load}(M', c)$ of messages in M' that must pass through c is no more than the capacity $\text{cap}(c)$, then no messages will be lost by concentrating the messages into c. We shall say that c is *congested by M'* if $\text{load}(M', c) > \text{cap}(c)$. We now show that the likelihood of channel congestion decreases exponentially with channel capacity if the probability of choosing a given message out of M is sufficiently small.

Lemma 4 *Let M be a set of messages on a fat-tree, let $\lambda(M)$ be the load factor on the fat-tree due to M, let M' be a p-subset of messages from M, and let c be a channel through which a given message $m \in M'$ must pass. Then the probability is at most $(ep\lambda(M))^{\text{cap}(c)}$ that channel c is congested by M'.*

Proof. Channel c is congested by M' if $\text{load}(M',c) > \text{cap}(c)$. There is already one message from the set M' going through channel c, so we must determine a bound on the probability that at least $\text{cap}(c)$ other messages go through c. Using Lemma 2 with $s = \text{cap}(c)$ and $t = \text{load}(M,c)$, the probability that the number of messages sent through channel c is greater than the capacity $\text{cap}(c)$ is less than

$$\begin{aligned} B(\text{cap}(c), \text{load}(M,c), p) &\leq \left(\frac{ep\,\text{load}(M,c)}{\text{cap}(c)}\right)^{\text{cap}(c)} \\ &\leq (ep\lambda(M))^{\text{cap}(c)} \ . \blacksquare \end{aligned}$$

The next lemma will analyze the probability that a given message of a p-subset of M gets delivered. In order to do the analysis, however, we must select p small enough so that it is likely that the message passes exclusively through uncongested channels. The choice of p depends on the capacities of channels in the fat-tree. For convenience, we define one parameter of the capacities which will enable us choose a suitable upper bound for p.

> **Definition:** The *congestion parameter* r of fat-tree is the smallest positive value such that for each simple path $c_1, c_2, \ldots, c_l$ of channels in the fat-tree, we have
>
> $$\sum_{k=1}^{l} \left(\frac{e}{r}\right)^{\text{cap}(c_k)} \leq \frac{1}{2} \ .$$

For a fat-tree based on a complete binary tree, the longest simple path is at most $2\lg n$, where n is the number of processors, and thus $r \leq 4e\lg n$. For universal fat-trees, the congestion parameter is a constant because the capacities of channels grow exponentially as we go up the tree. (All we really need is arithmetic growth in the channel capacities.) The congestion parameter is also constant for any fat-tree based on a complete binary tree if all the channels have capacity $\Omega(\lg\lg n)$. The remaining analysis treats the congestion parameter r as a constant, but the analysis does not change substantially for other cases.

We now present the lemma that analyzes the probability that a given message gets delivered.

> **Lemma 5** *Let M be a set of messages on a fat-tree with congestion parameter r, let $\lambda(M)$ be the load factor on the fat-tree due to M, and let m be an arbitrary message in M. Suppose M' is a p-subset of M, where $p \leq 1/r\lambda(M)$. Then if M' is sent, the probability that m gets delivered is at least $\frac{1}{2}p$.*

Proof. The probability that $m \in M$ is delivered is at least the probability that $m \in M'$ times the probability that m passes exclusively through uncongested channels. The probability that $m \in M'$ is p, and thus we need only show that, given $m \in M'$, the probability is at least $\frac{1}{2}$ that every channel through which m must pass is uncongested. Let $c_1, c_2, \ldots, c_l$ be the channels in the fat-tree through which m must pass. The probability that channel c_k is congested is less than $(e/r)^{\text{cap}(c_k)}$ by Lemma 4. The probability that at least one of the channels is congested is, therefore, much less than

$$\sum_{k=1}^{l} \left(\frac{e}{r}\right)^{\text{cap}(c_k)} \leq \frac{1}{2} \ ,$$

by definition of the congestion parameter. Thus, the probability that none of the channels are congested is at least $\frac{1}{2}$. $\blacksquare$

We now focus our attention on *RANDOM* itself. The next lemma analyzes the innermost loop (lines 3–6) of *RANDOM*'s subroutine *TRY-GUESS*. At this point in the algorithm, there is a set U of undelivered messages and a value for λ. The lemma shows that if λ is indeed an upper bound on the load factor $\lambda(U)$ of the undelivered messages when the loop begins, then $\lambda/2$ is an upper bound after the loop terminates. This lemma is the crucial step in showing that *RANDOM* works.

> **Lemma 6** *Let U be a set of messages on an n-processor fat-tree with congestion parameter r, and assume $\lambda(U) \leq \lambda$. Then after lines 3–6 of RANDOM's subroutine TRY-GUESS, the probability is at most $O(1/n^2)$ that $\lambda(U) > \frac{1}{2}\lambda$.*

Proof. The idea of the proof is to show that the load factor of an arbitrary channel c remains larger than $\frac{1}{2}\lambda$ with probability $O(1/n^3)$. Since the channel c is chosen arbitrarily out of the $4n-2$ channels in the fat-tree, the probability is at most $O(1/n^2)$ that any of the channels is left with load factor larger than $\frac{1}{2}\lambda$.

For convenience, let C be the subset of messages that must pass through channel c and are undelivered at the beginning of the innermost loop in *RANDOM*. Let $C_0 = C$, and for $i \geq 1$, let $C_i \subset C_{i-1}$ denote the set of undelivered messages at the end of the ith iteration of the loop. Notice that $\lambda(C_i, c) = |C_i|/\text{cap}(c)$, since $|C_i| = \text{load}(C_i, c)$.

We now show there exists values for the constants k_1 and k_2 in line 3 of *TRY-GUESS* such that for $z = \max\{k_1\lambda, k_2\lg n\}$, the probability is $O(1/n^3)$ that $\lambda(C_z, c) > \frac{1}{2}\lambda$, or equivalently, that

$$|C_z| > \frac{1}{2}\lambda\text{cap}(c) \ . \tag{1}$$

It suffices to prove that the probability is $O(1/n^3)$ that fewer than $\frac{1}{2}|C|$ messages from C are delivered during the z cycles under the assumption that $|C_i| > \frac{1}{2}\lambda\text{cap}(c)$ for $i = 0, 1, \ldots, z-1$. The intuition behind the assumption $|C_i| > \frac{1}{2}\lambda\text{cap}(c)$ is that otherwise, the load factor on channel c is already at most $\frac{1}{2}\lambda$ at this step of the iteration. The reason we need only bound the probability that fewer than $\frac{1}{2}|C|$ messages are delivered during the z cycles is that inequality (1) implies that the number of messages delivered is fewer than $|C| - \frac{1}{2}\lambda\text{cap}(c) \leq |C| - \frac{1}{2}\lambda(C,c)\text{cap}(c) \leq \frac{1}{2}|C|$.

We shall establish the $O(1/n^3)$ bound on the probability that at most $\frac{1}{2}|C|$ messages are delivered in two steps. For convenience, we shall call a cycle *good* if at least $\text{cap}(c)/8r$ messages are delivered, and *bad* otherwise. In the first step, we bound the probability that a given cycle is bad. Using Lemma 5 with $p = 1/r\lambda \leq 1/r\lambda(U) \leq 1/r\lambda(C_i)$ in conjunction with the assumption that $|C_i| > \frac{1}{2}\lambda\text{cap}(c)$, we can conclude that the expected number of messages delivered in any given cycle is greater than $\frac{1}{2r\lambda}\frac{1}{2}\lambda\text{cap}(c) \geq \text{cap}(c)/4r$. Then by Lemma 3, the probability that a a given cycle is bad is at most $1 - 1/(8r-1) < 1 - 1/8r$. (Although this bound is sufficiently strong to prove our theoretical results, it is weak because the probability that a message is delivered in a given cycle is not independent from the probabilities for other messages, and thus we must rely on the bound given by Lemma 3. In practice, one would anticipate that the dependencies are weak, and that the algorithm would be effective with much smaller values for the constants k_1 and k_2 than we can prove here.)

The second step bounds the probability that a substantial fraction of the z delivery cycles are bad. Specifically, we show that the probability is $1 - O(1/n^3)$ that at least some small constant fraction q of the z cycles are good. By picking $k_1 = 4r/q$, which implies $z \geq 4r\lambda/q$, at least $qz\text{cap}(c)/8r \geq \frac{1}{2}|C|$ messages will be delivered. We bound the probability that at least $(1-q)z$ of the z cycles are bad by using a counting argument. There are $\binom{z}{(1-q)z}$ ways of picking the bad cycles, and the probability that a cycle is bad is at most $1 - 1/8r$. Thus, the probability that at most $\frac{1}{2}|C|$ messages are delivered is

$$\begin{aligned}
&\Pr\left\{\leq \tfrac{1}{2}|C| \text{ messages delivered}\right\} \\
&\quad\leq \binom{z}{(1-q)z}\left(1-\frac{1}{8r}\right)^{(1-q)z} \\
&\quad\leq \left(q^q(1-q)^{1-q}\right)^{-z}\left(1-\frac{1}{8r}\right)^{(1-q)z} \\
&\quad\leq 2^{-z/12r} ,
\end{aligned}$$

if we choose $q = 1/e^4 r \ln r$, as the reader may verify. Since $z = \max\{k_1\lambda, k_2 \lg n\}$, if we choose $k_2 = 36r$, the probability that fewer than $\frac{1}{2}|C|$ messages are delivered is at most $1/n^3$. ∎

Now we can analyze *RANDOM* as a whole.

> **Theorem 7** *For any message set M on an n-processor fat-tree, the probability is at least $1 - O(1/n)$ that RANDOM will deliver all the messages of M within the number of delivery cycles specified by Figure 2.*

Proof. First, we will show that if $\lambda_{guess} \geq \lambda(M)$, the probability is at most $O(1/n)$ that the loop in lines 2 through 8 of *TRY-GUESS* fails to yield $\lambda(U) \leq 1$. Initially, $\lambda \geq \lambda(U)$, and we know from Lemma 6 that the probability is at most $O(1/n^2)$ that any given iteration of the loop fails to restore this condition as λ is halved. Since there are $\lg \lambda_{guess}$ iterations of the loop, we need only make the reasonable assumption that λ_{guess} is polynomial in n to obtain a probability of at most $O(1/n)$ that $\lambda(U)$ remains greater than 1 after all the iterations of the loop.

Now we just need to count the number of delivery cycles which will have been completed by the time we call *TRY-GUESS* with a λ_{guess} such that $\lambda(M) \leq \lambda_{guess}$. Let us denote by λ^*_{guess} the first λ_{guess} which satisfies this condition, and then break the analysis down into cases according to the value of $\lambda(M)$.

For $\lambda(M) \leq 1$, we do not actually even call *TRY-GUESS*. We need only count the one delivery cycle executed in line 1 of *RANDOM*.

For $1 < \lambda(M) \leq 2$, we need add only the $k_2 \lg n$ cycles executed when we call *TRY-GUESS*(2).

For $2 < \lambda(M) < (k_2/k_1)\lg n$, the number of delivery cycles involved in each execution of *TRY-GUESS* is $O(\lg \lambda_{guess} k_2 \lg n)$, since we perform $O(\lg \lambda_{guess})$ iterations of the loop in lines 2–8 of *TRY-GUESS*, each containing $k_2 \lg n$ iterations of the loop in lines 3–6. The value of λ^*_{guess} is at most $\lambda(M)^2$, so the total number of delivery cycles is

$$\begin{aligned}
&O(\lg n \lg \lambda(M)^2) + O(\lg n \lg \lambda(M)) + O(\lg n \lg \sqrt{\lambda(M)}) \\
&\qquad + O(\lg n \lg \sqrt[4]{\lambda(M)}) + \cdots + O(\lg n) \\
&= \sum_{0 \leq i \leq 1 + \lg\lg\lambda(M)} O(\lg n \lg(\lambda(M)^{2^{1-i}})) \\
&= \sum_{0 \leq i \leq 1 + \lg\lg\lambda(M)} O(2^{1-i} \lg n \lg(\lambda(M))) \\
&= O(\lg n \lg \lambda(M)) ,
\end{aligned}$$

since the series is geometric.

For $\lambda(M) > (k_2/k_1)\lg n$, the number of delivery cycles executed by the time we reach line 8 of *RANDOM* is $O(\lg n \lg\lg n)$ according to the preceding analysis, and then we must continue in the quest to reach λ^*_{guess}. If $\lambda(M) \leq (k_2/k_1)\lg n \lg\lg n$, then we need only add the number of delivery cycles involved in the single call

TRY-GUESS$((k_2/k_1)\lg n \lg\lg n)$. This additional number of delivery cycles is also $O(\lg n \lg\lg n)$, which is $O(\lg n \lg \lambda(M))$.

If $\lambda(M) > (k_2/k_1)\lg n \lg\lg n$, the number of delivery cycles executed before reaching line 8 is $O(\lg n \lg\lg n)$ as before, which is $O(\lambda(M))$. We must then add $O(\lambda_{guess})$ cycles for each call of *TRY-GUESS* in line 10. Since λ^*_{guess} is at most $2\lambda(M)$, the total additional number of delivery cycles is

$$\begin{aligned} & O(2\lambda(M)) + O(\lambda(M)) + O(\lambda(M)/2) + \cdots \\ & \qquad + O(\lg n \lg\lg n) \\ = & \sum_{0\le i\le t} O(2^{1-i}\lambda(M)) \\ = & O(\lambda(M)) \ , \end{aligned}$$

where $t = 1 + \lg(k_1\lambda(M)/k_2 \lg n \lg\lg n)$. The total number of delivery cycles is thus $O(\lambda(M))$. ∎

The $1 - O(1/n)$ bound on the probability that *RANDOM* delivers all the messages can be improved to $1 - O(1/n^k)$ for any constant k by choosing $k_2 = 12(k+2)r$, or by simply running the algorithm through more choices of λ_{guess}.

We can also use *RANDOM* to obtain a routing algorithm which guarantees to deliver all the messages in finite time with expected number of delivery cycles given in Figure 2. We simply interleave *RANDOM* with any routing strategy that guarantees to deliver at least one message in each delivery cycle. If the number of messages is bounded by some polynomial n^k, then we choose k_2 such that *RANDOM* works with probability $1 - O(1/n^k)$.

5 Further results

This section contains additional results relevant to routing on fat-trees. We first present an improved version of the universality theorem from [8]. Then we give two results on fat-tree routing that follow from the analysis of *RANDOM*. Finally, we show that there are message sets on which a greedy routing strategy fails to work well.

Universality

The performance of the routing algorithm *RANDOM* allows us to generalize the universality theorem from [8], which states that a universal fat-tree of a given volume can simulate any other routing network of equal volume with only a polylog factor increase in the time required. The original proof assumed the simulation of the routing was off-line. Our results show that the simulation can be carried out in the more interesting on-line context.

Theorem 8 *Let FT be a universal fat-tree of volume v on a set of n processors, and let R be an arbitrary routing network also of volume v on a set of n processors. Then there is an identification of processors in FT with the processors of R with the following property. Any message set M that can be delivered in time t by R can be delivered by FT in time $O((t+\lg\lg n)\lg^3 n)$ with probability $1-O(1/n)$.*

Sketch of proof. The proof follows that of [8]. The reader is referred to that paper for details. The routing network R of volume v is mapped to *FT* in such a way that any message set M that can be delivered in time t by R puts a load factor of at most $O(t\lg(n/v^{2/3}))$ on *FT*. By Theorem 7, the message set M can be delivered by *RANDOM* in $O(t\lg(n/v^{2/3}) + \lg n \lg\lg n)$ delivery cycles with high probability. Since each delivery cycle takes at most $O(\lg^2 n)$ time, the result follows. ∎

Remark. The delivery cycle time of the off-line fat-trees presented in [8] is $\Theta(\lg n)$. The on-line fat-trees described in Section 1 have a basic delivery cycle time of $\Theta(\lg^2 n)$ because the concentrator switches have logarithmic depth. We have discovered a simpler on-line fat-tree with delivery cycle time of $\Theta(\lg n)$, but unfortunately, the number of delivery cycles required by a *RANDOM*-like algorithm is increased by a factor of $\lg n$. It seems reasonable to look for fat-tree structures which save the factor of $\lg n$ in delivery cycle time without displacing it elsewhere.

Off-line routing

Our analysis for *RANDOM* has repercussions for the off-line routing case. Since we have shown that with high probability, the number of delivery cycles given by Figure 2 suffices to deliver a message set with load factor λ, there must exist off-line schedules using this many delivery cycles, which improves the bound of $O(\lambda \lg n)$ given in [8]. The previous off-line bound was proved by deterministically constructing a routing schedule that achieves the bound. Our better bound does not yield a deterministic construction of the routing schedule, but it does yield a probabilistic one.

Larger channel capacities

We can improve the results for on-line routing if each channel c in the fat-tree is sufficiently large, that is if $\text{cap}(c) = \Omega(\lg n)$ Specifically, we can deliver a message set M in $O(\lambda(M))$ delivery cycles with high probability, i.e., we can meet the lower bound to within a constant factor. The better bound is achieved by the algorithm *RANDOM-2* shown in Figure 5.

```
$z \leftarrow 1$
while $M \neq \emptyset$ do
    for each message $m \in M$, choose a random
    number $i_m \in \{1, 2, \ldots, z\}$
    for $i \leftarrow 1$ to $z$ do
        send all messages $m$ such that $i_m = i$
    endfor
    $z \leftarrow 2z$
endwhile
```

Figure 5: Algorithm *RANDOM-2* for routing in a fat-tree with channels of capacity $\Omega(\lg n)$.

> **Theorem 9** *For any message set M on an n-processor fat-tree with channels of capacity $\Omega(\lg n)$, the probability is at least $1 - O(1/n)$ that RANDOM-2 will deliver all the messages of M in $O(\lambda(M))$ delivery cycles, if $\lambda(M)$ is polynomially bounded.*

Proof. Let the lower bound on channel size be $a \lg n$, and let n^k be the polynomial bound on the load factor $\lambda(M)$. We consider only the pass of the algorithm when z first exceeds $e2^{(k+2)/a}\lambda(M)$. We ignore previous cycles for the analysis of message routing, except to note that the number of delivery cycles they require is $O(\lambda(M))$.

We first consider a single channel c within a single cycle i from among the z delivery cycles in the pass. Since each message has probability $1/z$ of being sent in cycle i, we can apply Lemma 4 with $p = 1/z$ to conclude that the probability that channel c is congested in cycle i is at most

$$\begin{aligned} \left(\frac{e\lambda(M)}{z}\right)^{\mathrm{cap}(c)} &\leq 2^{-\frac{k+2}{a}\mathrm{cap}(c)} \\ &\leq 2^{-(k+2)\lg n} \\ &= \frac{1}{n^{k+2}} . \end{aligned}$$

Since there are $O(n)$ channels, the probability that there exists a congested channel in cycle i is $O(1/n^{k+1})$. Finally, since there are $z \leq 2e2^{(k+2)/a}\lambda(M) = O(\lambda(M)) = O(n^k)$ cycles, the probability is $O(1/n)$ that there exists a congested channel in any delivery cycle of the pass. ▮

Greedy strategies

We have shown that there are no message sets on which *RANDOM* fails to work well. It is natural to wonder whether a simple greedy strategy of sending all undelivered messages on each delivery cycle, and letting them battle their ways through the switches, might be as effective. We can show that no greedy strategy works as well as *RANDOM*. Specifically, for any $\lambda > 1$, there is a message set with load factor λ which causes the greedy strategy to take $\Omega(\lambda \lg n)$ delivery cycles on an n-processor fat-tree. This lower-bound result is based on an idea originally due to Miller Maley of MIT.

```
while $M \neq \emptyset$ do
    send $M$
    $M \leftarrow M - \{$messages delivered$\}$
endwhile
```

Figure 6: Algorithm *GREEDY* for delivering a message set M.

Figure 5 shows the greedy algorithm. The code for *GREEDY* does not completely specify the behavior of message routing on a fat-tree because the switches have a choice as to which messages to drop when there is congestion. (The processors also have this choice, but we shall think of them as being switches as well.) In the analysis of *RANDOM*, we could presume that all messages in a channel were lost if the channel was congested. To completely specify the behavior of *GREEDY*, we must define the behavior of switches when channels are congested.

The lower bound for *GREEDY* covers a wide range of switch behaviors. Specifically, we assume the switches have the following properties.

1. Each switch is *greedy* in that it only drops messages if a channel is congested, and then only the minimum number necessary.
2. Each switch is *oblivious* in that decisions on which messages to drop are not based on any knowledge of the message set other than the presence or absence of messages on the switch's input lines.

We define the switches of a fat-tree to be *admissible* if they have these two properties. The conditions are satisfied, for example, by switches that drop excess messages at random, or by switches that favor one input channel over another. An admissible switch can even base decisions on its previous decisions, but it cannot predict the future or make decisions based on knowing what (or how many) messages it or other switches have dropped.

> **Theorem 10** *Consider an n-processor fat-tree with admissable switches, where the channel capacities grow at a rate α in the range $1 < \alpha < 2$. Then for any $\lambda > 1$, there exists a message set with load factor λ on which GREEDY requires $\Omega(\lambda \lg n)$ delivery cycles.*

Sketch of proof. We use an adversary argument and constructs a message set in which all messages are directed out the root. We first specify that the root switch is congested for $\Omega(\lambda)$ delivery cycles and demand to know what decisions the switch has made. Of the two subtrees of the root switch, we call the one which provides more than half the delivered messages the *favored* subtree. If both supply the same number, we pick one arbitrarily to be favored.

We then recursively design a message set for the unfavored side that has load factor λ, and put as many messages as possible on the favored side without exceeding a load factor of λ for the entire fat-tree. We design the message set in such a way as to be consistent with our specification that the root switch be congested for $\Omega(\lambda)$ delivery cycles. The crux of the construction is to ensure that as we go down the fat-tree following unfavored sides of switches, the messages delivered earlier will not uncongest the switches lower down. At each level of the fat-tree, we show that $\Omega(\lambda)$ delivery cycles are required. ▮

6 Concluding remarks

The analysis of the algorithm *RANDOM* gives reasonably tight asymptotic bounds on its performance, but the constant factors in the analysis are large. In practice, smaller constants probably suffice, but it is difficult to simulate the algorithm to determine what constants might be better. Unlike Valiant's algorithm for routing on the hypercube, our algorithm does not have the same probabilistic behavior on all sets of messages, and therefore, the simulation results may be highly correlated with the specific message sets chosen. The search for good constants is thus a multidimensional search in a large space, where each data point represents an expensive simulation.

The idea of using load factors to analyze arbitrary networks is a natural one. We have been successful in analyzing fat-trees using this measure of routing difficulty. It seems unlikely that large parallel supercomputers will only need to route permutations, but rather, they will need some distributed means to break apart their message sets into routable permutations. We expect that analysis in terms of load factor can be applied to other networks with positive results.

Acknowledgments

We have benefited tremendously from the helpful discussions and technical assistance of members of the theory of computation group at MIT. Thanks to Ravi Boppana, Thang Bui, Benny Chor, Peter Elias, Oded Goldreich, Johan Hastad, Alex Ishii, Tom Leighton, Bruce Maggs, Miller Maley, Cindy Phillips, Ron Rivest, and Peter Shor.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \log n$ parallel steps," *Combinatorica*, Vol. 3, No. 1, 1983, pp. 1–19.

[2] R. Aleliunas, "Randomized parallel communication," *Proceedings of the 1st Annual ACM Symposium on Principles of Distributed Computing*, August 1982, pp. 60–72.

[3] V. E. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic,* New York: Academic Press, 1965.

[4] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *Annals of Mathematical Statistics*, Vol. 23, 1952, pp. 493–507.

[5] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. 1, 2nd edition, New York: John Wiley & Sons, 1957.

[6] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *JACM,* Vol. 27, No. 4, October 1980, pp. 831–838.

[7] F. T. Leighton, *Complexity Issues in VLSI,* Cambridge, Massachusetts: MIT Press, 1983.

[8] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, October 1985, to appear.

[9] N. Pippenger, "Parallel communication with limited buffers," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science,* IEEE, October 1984, pp. 127–136.

[10] E. Upfal, "Efficient schemes for parallel communication," *JACM,* Vol. 31, No. 3, July 1984, pp. 507–517.

[11] L. G. Valiant, "A scheme for fast parallel communication," *SIAM Journal on Computing,* Vol. 11, No. 2, May 1982, pp. 350–361.

[12] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," *Proceedings of the 13th Annual ACM Symposium on Theory of Computing,* May 1981, pp. 263–277.

[13] A. Waksman, "A permutation network," *JACM,* Vol. 15, No. 1, January 1968, pp. 159–163.