Automatic Methods for Hiding Latency in High Bandwidth Networks (Extended Abstract)

Matthew Andrews* Tom Leighton† P. Takis Metaxas‡ Lisa Zhang§

Abstract

In this paper we describe methods for mitigating the degradation in performance caused by high latencies in parallel and distributed networks. Our approach is similar in spirit to the "complementary slackness" technique for latency hiding but has the advantage that the slackness does not need to be provided by the programmer and that large slowdowns are not needed in order to hide the latency. For example, given any 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 (which can be quite large in comparison to the average delay). In the case when 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 simulate an $n \times n$ array with unit link delays using slowdown $O(d_{\text{ave}}^{2/3} \log^{5/3} n)$ on an $(n^2 d_{\text{ave}}^{-2/3} \log^{-5/3} n)$-node array with average link delay d_{ave}.

We anticipate that our results will be of interest in the context of parallel and distributed computing on networks of workstations (NOWs). NOWs typically have the luxury of very high bandwidth links but suffer from latency problems caused by long wires or delays in accessing the network. Our work suggests an approach for overcoming such problems in a way that can be made transparent to the programmer (e.g., by making the network appear to function as if it were comprised of low-latency links).

*Department of Mathematics and Laboratory for Computer Science, MIT. Supported by NSF contract 9302476-CCR and ARPA contract N00014-95-1-1246. Email: andrews@math.mit.edu.

†Department of Mathematics and Laboratory for Computer Science, MIT. Supported by ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246. Email: ftl@math.mit.edu.

‡Department of Computer Science, Wellesley College. Supported by NSF contract 9504421-CCR and ARPA contract N00014-95-1-1246. Email: pmetaxas@wellesley.edu.

§Department of Mathematics and Laboratory for Computer Science, MIT. Supported by an NSF graduate fellowship and ARPA contract N00014-95-1-1246. Email: ylz@math.mit.edu.

1 Introduction

In this paper 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 communication 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 edge or link in the network has a delay which models the latency associated with using the edge. We also assume that the links can be pipelined (or that they have sufficient bandwidth) so that P "packets" can be sent across a link with delay d in $d + P - 1$ steps.

Most papers describing algorithms for parallel or distributed computation assume a model of computation in which all the edges 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, 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 networks of workstations (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).

Several methods have been devised in an attempt to deal with communication latencies. The simplest method is to slow down the computation to the point where the latency is accommodated. This approach is most commonly used at 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, in the CM-5 [10, 1] 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 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. Although this approach has worked well in some special examples [9, 5] we have found that is is not helpful in the scenarios we consider in this paper.

Perhaps the best and 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 [13] swapped between unrelated threads while waiting for the data. The CM-1 and CM-2 were designed to emulate much larger virtual machines so that a single processor would perform the computation of many virtual processors [14, 4]. The technique also forms a critical component of Valiant's bulk synchronous model of parallel computing [15, 16] and it has been employed in several algorithms papers [11, 7, 3, 12, 6].

In all of the preceding examples, however, it is incumbent on the programmer to provide the slackness or pipelining needed 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.

In this paper, we 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 communication 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.

For example, consider the problem of emulating an n-processor ring G with unit-delay links on an n-processor ring H with arbitrary delays on the links. *A priori,* it would seem that any such emulation would require slowdown $d_{\max}$ where $d_{\max}$ is the largest delay in H. (Indeed, this is the delay that would be incurred by prior approaches, although it should be pointed out that the prior approaches could preserve efficiency by using only $n/d_{\max}$ of the processors of H to carry out the emulation.) In the paper, however, we show how to accomplish the emulation with slowdown $O(\sqrt{d_{\text{ave}}})$, where d_{ave} is the average delay. This is optimal in some cases. (Efficiency is preserved here too since we use only $n/\sqrt{d_{\text{ave}}}$ processors to carry out the emulation.) The improvement is particularly impressive in the case when $d_{\max} \gg d_{\text{ave}}$, which is often the case in NOWs. More generally, we give a constant-times optimal approximation algorithm for minimizing the slowdown for any set of delays on the edges of H, and we also show how to carry out the emulation when H is an arbitrary bounded-degree fixed-connection network.

We also consider the problem of emulating an $n \times n$ array with unit-delay links on an $n \times n$ array with arbitrary delays. We consider both worst-case and random delay models. In the former case,[1] we show how to achieve slowdown $O\left(d_{\text{ave}}^{2/3} \log^{5/3} n\right)$. In the latter case we achieve slowdown $O\left(d_{\text{ave}}^{2/3}\right)$ with high probability, which is optimal in some cases. Both results are substantially more difficult than for the ring. Whether or not the $\log^{5/3} n$ factor can be removed from the worst-case upper bound remains as an interesting open question.

Our methods can also be used to embed arrays on other networks, but the resulting bounds on slowdown are far from tight. Indeed, the problem of embedding a unit-delay network G on a network H with arbitrary delays so as to minimize slowdown seems, in general, to be a very challenging problem.

In this paper the computation performed by processor p at time step t depends only on the computations performed by p and its neighbors at step $t-1$. This model is sufficiently general so as to be applicable for many algorithms, such as matrix operations, Fourier transforms, etc. However, it does not include applications where the computation performed by p depends on the state of a potentially large local memory or database. In [2] we consider a different model which handles this situation. In this new model we show how to simulate a linear array with unit delay links using slowdown $O\left(\sqrt{d_{\text{ave}}} \log^3 n\right)$ on an n-node network with average link delay d_{ave}.

The remainder of the paper is divided into sections as follows. In section 2 we present our results for rings. (In fact, the results of section 2 are described in terms of linear arrays, but since a linear array can simulate a ring with slowdown 2 the distinction is not important for our purposes.) In section 3 we present our results for $n \times n$ arrays in the worst-case and randomized models.

[1] Here we need to assume that the bandwidth available on the links of the host network is a $\Theta(\log^{3/2} n)$ factor greater than the bandwidth that is used on the links of the guest network. Otherwise, the slowdown may increase by a factor of $O(\log^{3/2} n)$. Bandwidth is not an issue in the randomized model.

2 Linear Arrays

2.1 Average Delay – An Upper Bound

Let the network G be an n-processor linear array with unit delay on all the edges. Let the network H be an n-processor linear array with arbitrary delays. Call these delays $d_1, d_2, \ldots, d_{n-1}$ respectively, where $d_i \geq 1$ is the delay on the ith edge of H. We wish to simulate network G on H. We refer to G as the *guest* and H as the *host*. The computation performed by processor g_i of G at time step t is denoted as pebble (i,t). In T steps G creates $n \times T$ pebbles, where pebble (i,t) depends on pebbles $(i-1,t-1)$, $(i,t-1)$ and $(i+1,t-1)$. In a *simulation* of G, H performs the same step-by-step computations as G when G is used in a general purpose way, i.e. H computes every pebble created by G. We first present algorithm SWEEP in which H simulates G with a slowdown of $s = O(d_{\text{ave}})$, where $d_{\text{ave}} = \sum_{k=1}^{n-1} d_k/(n-1)$ is the average delay of H.

Consider the first $n/2$ steps of computation 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 $i \leq t$. In algorithm SWEEP network H first simulates the bottom half of L and then the bottom half of R. At this point every pebble in the first $n/2$ steps of computation by G is simulated. Network H then proceeds to simulate the next $n/2$ steps 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 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 $t \leq n/2$. Notice 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, SWEEP transforms a process which involves two-way communication into a process which involves only one-way communication.

Lemma 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. Notice that the intersection of R and L is only computed once. Now SWEEP repeats to simulate the next $n/2$ steps. Therefore, the slowdown is upper bounded by,

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

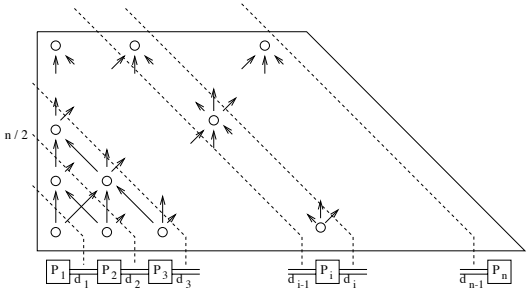

Figure 1: Algorithm SWEEP. Each slanted stripe is simulated by one processor of H. Arrows correspond to communications. Dotted lines correspond to the delays d_i encountered by communications.

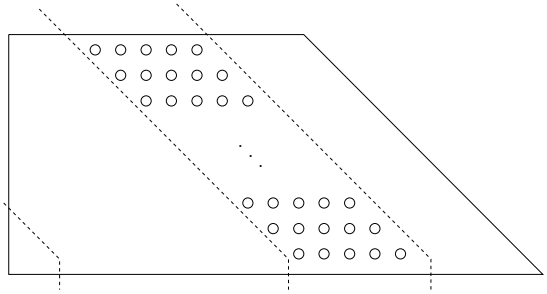

Figure 2: Algorithm FATSTRIPE. Processor p_i simulates stripe i which has width $\ell = n/m$. All the pebbles in stripe i are computed by time step $\ell n/2 + \sum_{k=1}^{i-1} d_k$.

2.2 Tight Upper and Lower Bounds

To get a better upper bound on the best achievable slowdown we introduce algorithm FATSTRIPE, in which a portion of network H (with a smaller average delay) is used. Suppose FATSTRIPE uses an interval of m processors to carry out the simulation. Without loss of generality, assume interval I 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, processor i computes every pebble in stripe i. (See Figure 2.)

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

Proof: It is obvious that p_1 finishes simulating stripe 1 at step $\ell(\ell+1)/2 \leq \ell n/2$. By an inductive argument similar to that for SWEEP one can verify that, after waiting for $\sum_{k=1}^{i-1} d_k$ steps, processor p_i $(2 \leq i \leq n)$ can compute the pebbles in stripe i one by one from left to right and from bottom to top with no more waiting. Since each stripe contains at most $\ell n/2$ pebbles, p_i finishes simulating stripe i by step $\ell n/2 + \sum_{k=1}^{i-1} d_k$. □

Hence, the slowdown is $s = O(n/m + \sum_{k=1}^{m-1} d_k/n)$. Algorithm FATSTRIPE can use any interval. In particular, FATSTRIPE finds 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 FATSTRIPE *achieves slowdown*

$$s = \min_{\text{intervals } I} O(n/m_I + d_I m_I/n).$$

In the special case when all the delays in H are d FATSTRIPE achieves a slowdown of $O(\sqrt{d})$ by using an interval of $n/\sqrt{d}$ processors. There must exist an interval I with $M_I = n/\sqrt{d_{\text{ave}}}$ processors and average delay $d_I \leq d_{\text{ave}}$. Hence,

Corollary 4 FATSTRIPE *achieves slowdown* $O(\sqrt{d_{\text{ave}}})$, *where* d_{ave} *is the average delay of* H.

We now show that the upper bound, $s = \min_{\text{intervals } I} O(n/m_I + d_I m_I/n)$, in Theorem 3 is asymptotically tight by showing that $\min_{\text{intervals } I} \max\{n/2m_I, d_I m_I/2n\}$ is a lower bound on the best achievable slowdown even if we allow redundant computation.

Lemma 5 *The top pebble,* $(1, n)$ *of triangle* L, *cannot be computed at a time step earlier than*

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

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 q' be a processor that has computed $(1, n-1)$ *and* has passed this information to q, and q'' be a processor that has computed $(2, n-1)$ *and* has passed this information to q. (Notice that other processors may compute $(1, n-1)$ and $(2, n-1)$. We are only concerned with processors which pass information to q.) Now let $q'^{(1,n-1)}$ and $q''^{(2,n-1)}$ be the children of $q^{(1,n)}$ in T. We proceed to construct the childrenfor $q'^{(1,n-1)}$ and $q''^{(2,n-1)}$. 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 $\hat{I}$ be the smallest interval that contains all the processors appearing in tree T. If processors x and y are at the two ends of $\hat{I}$, 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 $d_{\hat{I}} m_{\hat{I}}/2$ steps. It follows that pebble $(1, n)$ cannot be computed at a step earlier than $d_{\hat{I}} m_{\hat{I}}/2$. By a work argument, $(1, n)$ cannot be computed before step $n^2/2m_{\hat{I}}$. Hence, $(1, n)$ cannot be computed at a step earlier than $\min_{\text{intervals } I} \max\{n^2/2m_I, d_I m_I/2\}$. □

It follows that the slowdown in simulating triangle L is lower bounded by $\min_{\text{intervals } I} \max\{n^2/2m_I, d_I m_I/2\}$. By a similar argument to Lemma 5 none of the pebbles (i, n), for $1 \leq i \leq n$, can be computed earlier than $\min_{\text{intervals } I} \max\{n^2/2m_I, d_I m_I/2\}$. By repeating this construction the first kn steps of G cannot be simulated in time less than $k \min_{\text{intervals } I} \max\{n^2/2m_I, d_I m_I/2\}$. Therefore, we obtain,

Theorem 6 *The slowdown of any simulation of* G *by* H *is lower bounded by*

$$\min_{\text{intervals } I} \Theta(n/m_I + d_I m_I/n).$$

Hence, FATSTRIPE *is optimal up to a constant factor.*

2.3 Simulating Linear Arrays with General Networks

We now consider simulating a linear array G on a general n-node network N with average delay d_{ave}. The following fact [8] is used in our argument.

Fact 7 *An* n*-node linear array can be one-to-one embedded with dilation* 3 *in any connected* n*-node network.*

Let $\bar{N}$ be the n-node linear array embedded in N by Fact 7. The proof of Fact 3 [8, page 470] implies that if N has bounded degree δ then $\bar{N}$ has average delay at most δd_{ave}. By Corollary 4, $\bar{N}$ can simulate G with a slowdown of $O(\sqrt{\delta d_{\text{ave}}})$. Thus,

Theorem 8 *A bounded degree network* N *with average delay* d_{ave} *can simulate an* n*-node linear array* G *with a slowdown of* $O(\sqrt{d_{\text{ave}}})$.

Theorem 8 does not hold when N has unbounded degree. Consider the following example. Let N 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 2 adjacent cliques has delay n, then N has $d_{\text{ave}} < 4$. Suppose m connected cliques are used to simulate n steps of G. A work argument implies a slowdown of at least $\sqrt{n}/m$. A linear array embedded in these m connected cliques has a total delay of at least mn, which implies a slowdown of m. Hence, the slowdown is at least $\max\{\sqrt{n}/m, m\} \geq n^{1/4}$, whereas the average delay is a constant.

Notice that all the results in this section apply to rings, since a linear array can simulate a ring with a slowdown of 2 [8].

3 Two Dimensional Arrays

3.1 The 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 pebble created by processor $g_{i,j}$ of G at time step t is labeled (i, j, t).

Algorithm 2D-SWEEP is a 2-dimensional analogue of SWEEP. Let P_1 be the pyramid with vertices $(1, 1, 1)$, $(1, n, 1)$, $(n, 1, 1)$, $(n, n, 1)$ and $(1, 1, n)$. Pyramids P_2, P_3 and P_4 are defined similarly. To simulate the first $n/2$ steps of G 2D-SWEEP simulates P_1, P_2, P_3 and P_4. At this point every pebble in the first $n/2$ steps of G is simulated. 2D-SWEEP then proceeds to simulate the next $n/2$ steps in a similar manner. 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 $1 \leq t \leq \min\{i, j\}$. (Algorithm 2D-SWEEP only needs to simulate the bottom half of P_1. However,

for simplicity we assume 2D-SWEEP simulates all of P_1. The asymptotic performance remains the same.) 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. (Monotone paths travel in two directions, up and right.)

Lemma 9 *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 case 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 3.) 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 $\max\{D_{i-1,j}+x_{i-1,j}, D_{i,j-1}+y_{i,j-1}\}+t = D_{i,j}+t$. □

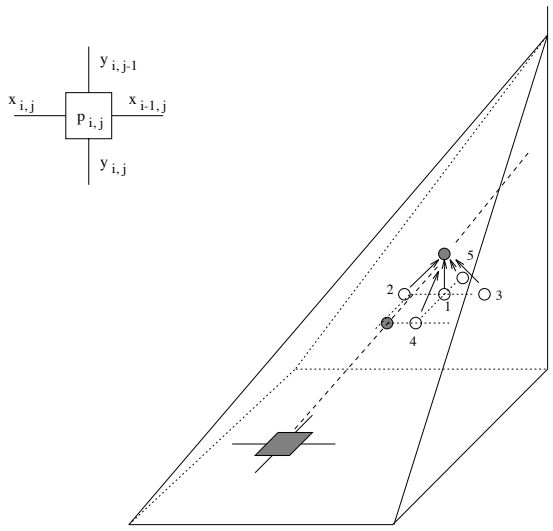

Figure 3: Algorithm 2D-SWEEP. Pyramid P_1. The dashed line represents the ray of pebbles computed by processor $p_{i,j}$ (which is shown in the upper left corner). Two of those pebbles, computed at times t and $t-1$, are shown shaded. The five numbered pebbles are those that $(i-t+1, j-t+1, t)$ depends on.

Hence, 2D-SWEEP simulates pyramid P_1 in $D_{n,n}+n$ steps. The slowdown is therefore $O(D_{n,n}/n)$. Let d_{ave} be 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 now introduce algorithm FATRAY, a two dimensional analogue of FATSTRIPE, to achieve an upper bound on the slowdown which is better than $O(D_{n,n}/n)$. Suppose FATRAY uses processors $p_{i,j}$ $(1 \le i,j \le m)$ of H to carry out the simulation. Pyramid P_1 is divided into m^2 rays, each of which has size $\ell \times \ell = \frac{n}{m} \times \frac{n}{m}$. Again, $p_{i,j}$ computes every pebble in ray $R_{i,j}$. By an inductive argument similar to that for FATSTRIPE and 2D-SWEEP one can verify that after waiting for $D_{i,j}$ steps processor $p_{i,j}$ is able to compute all the pebbles in $R_{i,j}$ one by one with no more waiting. (Processor $p_{i,j}$ first computes all the pebbles on the bottom plane of $R_{i,j}$ and then moves up.) Since each ray contains at most $\ell^2 n$ pebbles, $p_{i,j}$ finishes simulating ray $R_{i,j}$ by step $\ell^2 n + D_{i,j}$. This implies that the slowdown is $O(n^2/m^2 + D_{m,m}/n)$. Let S be a contiguous subsquare of size $m_S \times m_S$, and let D_S be the length of the longest monotone path in S. By finding a contiguous subsquare S that minimizes $n^2/m_S^2 + D_S/n$, we obtain,

Theorem 10 FATRAY *achieves slowdown of*

$$\min_{\text{subsquares } S} O(n^2/m_S^2 + D_S/n).$$

Unfortunately, the slowdown can still be big compared with d_{ave}. For example, when H has n edges with delay n and they are spread out evenly in the network, the slowdown is $\min_{\text{subsquares } S} O(n^2/m_S^2 + D_S/n) = \Omega(n^{1/3})$ 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 11 *In the case where all the delays in H are d,* FATRAY *achieves a slowdown of $O(d^{2/3})$. This slowdown is optimal up to a constant factor.*

Proof: Theorem 10 implies a slowdown of $O(d^{2/3})$ when FATRAY uses a subsquare of size $\frac{n}{d^{1/3}} \times \frac{n}{d^{1/3}}$. We now show that the slowdown is asymptotically tight. 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 $1 \le t < d^{1/3}$, 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 $|A| = \Omega\left(\left(d^{1/3}\right)^3\right) = \Omega(d)$ 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. □

The proof of Theorem 11 can be generalized to a k-dimensional array. Suppose network 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

Theorem 12 *Network H' can simulate G' with slowdown $O(d^{k/k+1})$ and this bound is tight.*

3.2 Worst Case Model

We now show that for any arrangement of the delays in H, we can embed G into H with a constant load such that the image in H of any monotone path in G has length $O(d_{\text{ave}} n \log^{5/2} n)$. This will then enable us to achieve slowdown $O(d_{\text{ave}} \log^{5/2} n)$. By applying Theorem 10 and choosing S appropriately we can then improve the slowdown to $O(d_{\text{ave}}^{2/3} \log^{5/3} n)$ using only $n^2 d_{\text{ave}}^{-2/3} \log^{-5/3} n$ processors.

Killing Useless Processors

We first build a 4-ary tree, T, to represent network H. Each node of T corresponds to a subsquare of H. In particular, the root represents the entire $n \times n$ square. The four children of the root represent the four $\frac{n}{2} \times \frac{n}{2}$

subsquares, etc. The leaves of T represent the individual processors of H. Tree T has depth $\log n$. It is easy to see the one-to-one correspondence between the 4^k depth k nodes of T and the depth k subsquares of H. A depth k subsquare has size $\frac{n}{2^k} \times \frac{n}{2^k}$. (See Figure 4.)

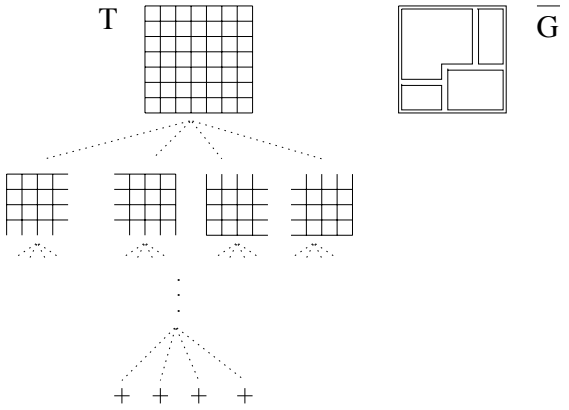

Figure 4: (Left) The 4-ary tree T that represents network H. (Right) The four depth 1 regions of $\bar{G}$.

Some processors may be unusable when they are surrounded by too much delay. We describe a 2-stage procedure to *kill* such processors. When a processor is killed, its corresponding leaf in T is removed.

Stage 1 It is clear that each processor p is contained in exactly one depth k subsquare. Call this subsquare S_p^k. Processor p is killed if the row or column of S_p^k that contains p has total delay more than $(c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right)$, for some constant c to be specified later.

Lemma 13 *At most $2n^2/c$ processors are killed in stage 1.*

Proof: The total delay of H is $2n^2 d_{\text{ave}}$. It follows that the number of depth k rows and columns with delay more than $(c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right)$ is at most $\frac{2n2^k}{c \log n}$. Since each depth k row/column contains $\frac{n}{2^k}$ processors, the number of processors killed at depth k is at most $\frac{2n^2}{c \log n}$. □

Stage 2 A subsquare may not be useful if most of its processors have been killed. Hence, if a depth k subsquare contains fewer than $\frac{1}{c \log n} \frac{n^2}{4^k}$ live processors, we kill off its remaining processors. Moreover, we also kill off processors so that the number of remaining live processors in any depth k square is an integer multiple of $\frac{1}{c \log n} \frac{n^2}{4^k}$. Since there are 4^k depth k subsquares, at most $\frac{n^2}{c \log n}$ processors are killed at depth k. Therefore,

Lemma 14 *At most n^2/c processors are killed at stage 2.*

By Lemmas 13 and 14, we know that at most $3n^2/c$ processors of H are killed. Hence H has $c_1 n^2$ live processors in total, where $c_1 \geq 1 - (3/c)$. We now mark each node of T with the number of leaves under it. (Notice that this number is the same as the number of live processors in the corresponding subsquare. The root is marked with the number $c_1 n^2$.)

The Embedding

Let network $\bar{G}$ be a $\sqrt{c_1} n \times \sqrt{c_1} n$ 2-dimensional array with unit delay edges only. It is easy to simulate G by $\bar{G}$ with constant slowdown. We describe algorithm EMBED that embeds the processors of guest network $\bar{G}$ one-to-one to the live processors of H. Algorithm EMBED makes sure that the image in H of any monotone path in $\bar{G}$ has length $O(d_{\text{ave}} n \log^{5/2} n)$. EMBED partitions $\bar{G}$ into *regions* recursively. At the kth level of recursion, EMBED establishes a one-to-one correspondence between the depth k regions of $\bar{G}$, the depth k nodes of T and the the depth k subsquares of H.

At the first level of recursion $\bar{G}$ is viewed as a depth 1 *grid* of size $2\sqrt{(c-3)\log n} \times 2\sqrt{(c-3)\log n}$. Each square in the depth 1 grid contains $\frac{1}{c \log n} \frac{n^2}{4}$ processors of $\bar{G}$. In tree T let a_1, a_2, a_3 and a_4 be the children of the root. Suppose the children are marked with numbers z_1, z_2, z_3 and z_4 respectively, then $z_1 + \ldots + z_4 = c_1 n^2$ and each of $z_1, \ldots, z_4$ is a multiple of $\frac{1}{c \log n} \frac{n^2}{4}$ by assumption. EMBED partitions $\bar{G}$ into four contiguous depth 1 regions $R_1, \ldots, R_4$. Each R_i $(1 \leq i \leq 4)$ contains z_i processors of $\bar{G}$, and corresponds to node a_i of T. Also, each R_i is made up from a subset of the squares that form the depth 1 grid. Notice that some region R_i may be empty, which corresponds to z_i being zero. The boundaries of these depth 1 regions form the depth 1 *boundary* in $\bar{G}$. EMBED then partitions each R_i into four depth 2 regions.

Suppose R is a depth $k-1$ region of $\bar{G}$, which corresponds to a depth $k-1$ node r in T. (Notice that R also corresponds to a depth $k-1$ subsquare of H.) By the construction of EMBED, the number of processors contained in region R is the same as the number marked on node r. At the kth level of recursion, $\bar{G}$ is viewed as a depth k grid of size $2^k\sqrt{(c-3)\log n} \times 2^k\sqrt{(c-3)\log n}$. Each square in the depth k grid contains $\frac{1}{c \log n} \frac{n^2}{4^k}$ processors of $\bar{G}$. EMBED partitions R into four contiguous depth k regions, each of which is made up from a subset of the squares that form the depth k grid. Also, these four depth k regions have sizes equal to the numbers marked on the children of r and therefore correspond to those nodes in T. The boundaries of these depth k regions form the depth k boundary in $\bar{G}$.

The depth $\log n$ regions are made up from individual processors in $\bar{G}$, which correspond to the leaves of T. Thus, we have a one-to-one embedding from the processors of $\bar{G}$ to the live processors of H.

We are now ready to bound the total delay on the image of a monotone path. Notice that the image in H of a monotone path in $\bar{G}$ is not necessarily monotone. The intuition for the argument is as follows. Whenever a monotone path crosses a depth k boundary in $\bar{G}$ for some k, some delay is picked up in H. However, by the construction of EMBED the boundaries in $\bar{G}$ are not too dense, and so a monotone path cannot cross the boundaries too many times.

Theorem 15 *Let P be a monotone path in $\bar{G}$. The image of P under the embedding of $\bar{G}$ in H has total delay $O(d_{\text{ave}} n \log^{5/2} n)$.*

Proof: Consider two consecutive processors a and b of $\bar{G}$ on path P. Let a_H and b_H be the corresponding processors in H. We want to bound the shortest

path distance between a_H and b_H. Choose the largest k such that a and b are in the same depth k region, then a_H and b_H are in the same depth k subsquare of H. After processors of H are killed in stage 1 the total delay between any two live processors in a depth k subsquare is at most $(2c\log n)\left(\frac{n}{2^k}d_{\text{ave}}\right)$. Hence, the distance between a_H and b_H is at most $(2c\log n)\left(\frac{n}{2^k}d_{\text{ave}}\right)$. Notice that in traveling from a to b path P crosses the depth ℓ boundary of G if and only if $\ell > k$. Hence, the total delay from a to b in the embedding is at most $\sum_{\ell:\text{ depth }\ell\text{boundary crossed}}(2c\log n)\left(\frac{n}{2^\ell}d_{\text{ave}}\right)$. It follows that the total delay on P under the embedding is at most $\sum_\ell(2c\log n)\left(\frac{n}{2^\ell}d_{\text{ave}}\right)C_\ell$, where C_ℓ is the number of times the depth ℓ boundary is crossed by P. By the definition of depth ℓ boundary from the depth ℓ grid, monotone path P can cross it at most $2^{\ell+1}\sqrt{(c-3)\log n}$ times, i.e. $C_\ell \le 2^{\ell+1}\sqrt{(c-3)\log n}$. Hence, the image of P in H under the embedding has total delay at most $\sum_{\ell=1}^{\log n}(2c\log n)\left(\frac{n}{2^\ell}d_{\text{ave}}\right)\left(2^{\ell+1}\sqrt{(c-3)\log n}\right) = O(d_{\text{ave}}n\log^{5/2} n)$. □

It is clear that the above argument holds for any constant $c > 3$. Hence, we can embed G into H with a constant load such that the image in H of any monotone path in G has length $O(d_{\text{ave}}n\log^{5/2} n)$. Network H can therefore simulate G with a slowdown of $O(d_{\text{ave}}\log^{5/2} n)$.

The preceding analysis focuses entirely on the issue of latency and ignores bandwidth constraints. This does not present any problems if the host array has sufficiently high bandwidth links. 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 $O(\log^{3/2} n)$. To see why, notice that each depth k region R is mapped to a depth k subsquare S of H with the same number of live processors. Moreover, if s is the number of live rows and columns of S, the number of connections from S to the rest of the network is at most $O(s\sqrt{\log n})$. Given an arbitrary routing problem of size s in which each request is on the boundary, subsquare S can handle this problem with $O(1)$ congestion. Using this fact we connect up the processors by recursively routing to the boundaries of subsquares. This means that the congestion induced at each depth is at most $O(\sqrt{\log n})$. Since each of the depths uses the same underlying edges it follows that the overall congestion is at most $O(\log^{3/2} n)$. Details of the proof are left for the full version.

We suspect that our bounds can be improved by a $\Theta(\sqrt{\log n})$ factor in congestion and a $\Theta(\log^{1/3} n)$ factor in latency-based slowdown by a more sophisticated method for embedding G into H, but as yet we have no proof.

Congestion problems are not an issue for the simulation algorithm in the next section (in the case when the latencies are randomly organized) since the communication paths in the simulation can be made to have constant length without loss of generality.

We adopt the grouping idea of FatRay and use an $m \times m$ contiguous subsquare of H for simulation. The quantity n^2/m^2 lower bounds the slowdown because of a work argument. Since we can always find an $m \times m$ subsquare of average delay at most d_{ave}, Theorem 15 implies a slowdown of $O\left((d_{\text{ave}}m\log^{5/2} m)/n\right)$. When $m = nd_{\text{ave}}^{-1/3}\log^{-5/6} n$, we have,

Theorem 16 *Network H with average delay d_{ave} can efficiently simulate G with a slowdown of $O(d_{\text{ave}}^{2/3}\log^{5/3} n)$.*

3.3 Randomized Model

Given any set of n^2 delays with average delay d_{ave} we show that the longest monotone path has length $D_{n,n} = O(nd_{\text{ave}})$ for most of the $(n^2)!$ permutations of the delays. That is, in the uniform distribution of the $(n^2)!$ permutations $D_{n,n} = O(nd_{\text{ave}})$ with high probability, and therefore the slowdown is $O(d_{\text{ave}})$ with high probability.

Constant Average Delay

We discuss the case in which the average delay is 2. For clarity of presentation we first assume that H has only two delays, 1 and $d > 1$. We then generalize the proof technique to arbitrary delays. Without loss of generality assume H has $\frac{n^2}{d}$ d-delays. Hence, the probability of an edge to have delay d is $1/d$. Notice that the probabilities are *not* independent, since the arrangement of the delays is a permutation of a given set of delays. However, we have the following useful fact.

Fact 17 *If e_1 and e_2 are two distinct edges, then*

$$\begin{aligned} &\Pr[\ e_1 \textit{ has delay } d \mid e_2 \textit{ has delay } d\ ] \\ \le\ &\Pr[\ e_1 \textit{ has delay } d\ ]. \end{aligned}$$

If a d-delay edge is isolated then we can route around it and replace it with a shortcut of small delay. The intuition is that in a random permutation most d-delays can be shortcut. We label an edge e *long* if the delay on e cannot be replaced by a shortcut of length less than 10; we label edge e *short* otherwise. In general, we have the following.

Lemma 18 *Let $a_1,\ldots,a_i$ be a set of labels from $\{$ long, short $\}^i$, and let X be the event that edge e_j, for $1 \le j \le i$, is labeled with a_j. We have,*

$$\Pr[\ \textit{Edge } e \textit{ long} \mid X\ ] \le 1/d^4,$$

for $i = o(n^2)$.

Proof: For clarity of presentation we only prove

$$\Pr[\ \text{Edge } e \text{ long} \mid X\ ] \le c/d^2,$$

for some constant c. Notice that when a 1×1 square contains only one d-delay, then this delay can be replaced by a shortcut of length 3. Consider the 1×1 squares that contain edges e, $e_1,\ldots,e_i$, and call them b, $b_1,\ldots,b_i$ respectively.

$$\begin{aligned} &\Pr[\ \text{Edge } e \text{ long} \mid X\ ] \\ \le\ &\Pr[\ \text{Edge } e \text{ long} \mid e_1,\ldots,e_i \text{ labeled short}\ ] \\ \le\ &\Pr[\ \text{Square } b \text{ contains at least 2 } d\text{-delays} \\ &\qquad \mid b_1,\ldots,b_i \text{ contain 1-delays only}\ ] \\ \le\ &\binom{4}{2}\left(\frac{n^2/d}{n^2-4i}\right)\left(\frac{n^2/d-1}{n^2-4i-1}\right) \\ \le\ &c/d^2, \end{aligned}$$

for some constant c. The first two inequalities follow from Fact 17. The last inequality follows from $i = o(n^2)$. The proof for

$$\Pr[\text{ Edge } e \text{ long } | \ e_1, \ldots, e_i \text{ long or short }] \leq 1/d^4$$

uses the same idea. Instead of a 1×1 square, we consider a 3×3 square whose middle 1×1 square contains edge e. Details are left for the full version. □

If d is large, Lemma 18 implies that with high probability every edge is short. When d is small, we use a moment generating function argument to show that every monotone path has only "a few" long edges. We have,

Theorem 19 *If the edges of H consist of $\frac{n^2}{d}$ d-delays and $n^2 - \frac{n^2}{d}$ 1-delays, then with probability $1-n^{-1}$ every monotone path in H has total delay less than cn, for a sufficiently large constant c.*

Proof: There are two cases to discuss.

Case 1: When $d > n/\log n$ we show that with high probability all edges are short. It follows from Lemma 18 that for any edge e, $\Pr[$ Edge e is long $] \leq 1/d^4 \leq \log^4 n/n^4 \leq \frac{1}{2}n^{-3}$. By a union bound, $\Pr[$ Every edge is short $] \geq 1-(2n^2)\left(\frac{1}{2}n^{-3}\right) \geq 1-n^{-1}$. Hence, with probability $1 - n^{-1}$ every monotone path has total delay at most $20n$, since the delay on a short edge is at most 10.

Case 2: When $d \leq n/\log n$ we use a moment generating function argument. Divide H into $m^2 = n^2/d^2$ boxes of size $d \times d$. Fix a monotone path P, and let X_i be the random variable that counts the number of long edges in box i on path P. We wish to upper bound $X = X_1 + \ldots + X_m$. We first compute $\Pr[\, X_i = k_i | X_1 = k_1, \ldots, X_{i-1} = k_{i-1} \,]$. Since there are at most $2d^2m \leq 2n^2/\log n = o(n^2)$ edges in boxes $1, \ldots, i$ on path P, we apply Lemma 18 to obtain,

$$\begin{aligned} P_i &= \Pr[\, X_i = k_i \mid X_1 = k_1, \ldots, X_{i-1} = k_{i-1} \,] \\ &\leq \binom{d^2}{k_i}\left(\frac{1}{d^4}\right)^{k_i} \leq \left(\frac{d^2 e}{k_i}\right)^{k_i}\left(\frac{1}{d^4}\right)^{k_i} \leq 1/2^{k_i}. \end{aligned}$$

Let $\lambda = 0.6 < \ln 2$ and $\delta = -\ln(1 - e^{\lambda}/2)$. The expectation of $e^{\lambda X}$ is,

$$\begin{aligned} E[e^{\lambda X}] &= E[e^{\lambda(X_1+\ldots+X_m)}] \\ &= \sum_{k\geq 0}\sum_{k_1+\ldots+k_m=k}\prod_{i=1}^{i=m} P_i e^{\lambda k_i} \\ &\leq \sum_{k\geq 0}\sum_{k_1+\ldots+k_m=k}\prod_{i=1}^{i=m} e^{\lambda k_i}/2^{k_i} \\ &\leq \left(\sum_{k\geq 0}(e^{\lambda}/2)^k\right)^m \\ &\leq \left(1 - e^{\lambda}/2\right)^{-m} = e^{\delta m}. \end{aligned}$$

By Markov's inequality we know that,

$$\begin{aligned} \Pr[\, X \geq \beta m] &= \Pr\left[\, e^{\lambda X} \geq e^{\lambda\beta m}\right] \\ &\leq \frac{E[e^{\lambda X}]}{e^{\lambda\beta m}} \leq e^{-(\lambda\beta-\delta)m}. \end{aligned}$$

Therefore,

$$\begin{aligned} &\Pr[\text{ The total delay on } P \text{ is more than } \beta n + 20n \,] \\ \leq\ &\Pr[\, X \geq \beta m \,] \leq e^{-(\lambda\beta-\delta)m}, \end{aligned}$$

since the delay on a short edge is at most 10. A monotone path goes through at most $2m$ $d \times d$ boxes. By a union bound we obtain that the probability that every monotone path has total delay less than $\beta n + 20n$ is at least $1 - 2^{2m}e^{-(\lambda\beta-\delta)m} \geq 1 - e^{-(\lambda\beta-\delta-2)m}$. Since $d \leq n/\log n$, i.e. $m \geq \log n$, this probability is at least $1 - n^{-1}$ for $\beta = 10$. (Notice that 2^{2m} does not upper bound the number of monotone paths, but rather the number of combinations of $2m$ $d \times d$ boxes that a monotone path can go through.) □

We now consider network H which has average delay 2 and arbitrary delays on its edges. Divide the delays of H into levels, where level ℓ ($0 \leq \ell \leq 2\log n$) consists of delays which are at least 2^{ℓ} but less than $2^{\ell+1}$. Let r_ℓ be the number of delays in levels $\ell' \geq \ell$. We perform an all-pairs shortest path algorithm on H^2 (using delays as the edge lengths), and replace each delay by the shortest path length. Let $\bar{H}$ be the network after short-cutting H. The following is analogous to Lemma 18.

Lemma 20 *Given fixed delays on edges $e_1, \ldots, e_i$ in $\bar{H}$, where $i = o(n^2)$, the probability that edge e of $\bar{H}$ has a level ℓ delay is at most $\left(\frac{r_{\ell-3}}{n^2}\right)^4$.*

Proof: For clarity of presentation we prove that the probability is at most $c\left(\frac{r_{\ell-2}}{n^2}\right)^2$, for some constant c. Consider the 1×1 squares that contain edges e, $e_1, \ldots, e_i$, and call them $b, b_1, \ldots, b_i$ respectively.

$$\begin{aligned} &\Pr[\text{ Edge } e \text{ has a level } \ell \text{ delay in } \bar{H} \\ &\qquad | \text{ Delays fixed on } e_1, \ldots, e_i \text{ in } \bar{H} \,] \\ \leq\ &\Pr[\text{ Square } b \text{ contains at least 2 delays from} \\ &\qquad \text{levels } \ell' \geq \ell - 2 \text{ in } H \\ &\qquad | \text{ Squares } b_1, \ldots, b_i \text{ have no delays from} \\ &\qquad \text{levels } \ell' \geq \ell - 2 \text{ in } H \,] \\ \leq\ &\binom{4}{2}\left(\frac{r_{\ell-2}}{n^2-4i}\right)\left(\frac{r_{\ell-2}-1}{n^2-4i-1}\right) \\ \leq\ &c\left(\frac{r_{\ell-2}}{n^2}\right)^2, \end{aligned}$$

for some constant c. The first inequality holds for a similar reason to Fact 17. The last inequality follows from $i = o(n^2)$.

The proof to derive the probability of $\left(\frac{r_{\ell-3}}{n^2}\right)^4$ uses the same idea. Instead of a 1×1 square, we consider a 3×3 square whose middle 1×1 square contains edge e. Details are left for the full version. □

For level ℓ, where $r_{\ell-3}$ is small, Lemma 20 implies that with high probability such big delays do not exist in $\bar{H}$. For level ℓ, where $r_{\ell-3}$ is big, we use a moment generating function argument, similar to the proof of Theorem 19, to show that every monotone path only has "a few" level ℓ delays. We have,

Theorem 21 *Suppose H is a network of average delay 2, and $\bar{H}$ is the network after short-cutting H. Then with probability $1 - n^{-1}$ every monotone path in $\bar{H}$ has delay less than cn, for a sufficiently large constant c.*

[2] If congestion is an issue then we would use shortest paths with a constant number of edges.

Proof: The proof uses the idea of Theorem 19. Let A be the event that the total delay from levels $\ell \geq 3$ on any monotone path is less than $64\beta n$ in $\bar{H}$, and B be the event that edges with level ℓ delays do not exist in $\bar{H}$, where $\ell \geq 3$ and $r_{\ell-3} < n \log n$. Since $\Pr[\, A \,] \geq \Pr[\, A \mid B \,] \Pr[\, B \,]$, we show that A happens with high probability by showing that both $\Pr[\, A \mid B \,]$ and $\Pr[\, B \,]$ are large.

From Lemma 20 we have,

$$\begin{aligned} & \Pr[\text{ Edge } e \text{ has a level } \ell \text{ delay, where } \ell \geq 3 \text{ and } r_{\ell-3} < n \log n \,] \\ \leq & \sum_{\ell:\ell\geq 3, r_{\ell-3}<n\log n} r_{\ell-3}^4/n^8 \leq 2\log n(\log^4 n/n^4) \\ \leq & \frac{1}{2}n^{-3}. \end{aligned}$$

By a union bound the probability that B happens is at least $1 - n^{-1}$.

We now upper bound $\Pr\left[\, \bar{A} \mid B \,\right]$ using case 2 of Theorem 19. For each level ℓ, where $\ell \geq 3$, let $m_{\ell-3} = r_{\ell-3}/n \geq \log n$.

$$\begin{aligned} & \Pr\left[\, \bar{A} \mid B \,\right] \\ = & \Pr[\text{ Total delay from levels } \ell \geq 3 \text{ is at least } 64\beta n \text{ on some monotone path} \mid B \,] \\ \leq & \Pr[\text{ Total delay from levels } \ell \geq 3 \text{ is at least } \beta \textstyle\sum_{\ell\geq 3} m_{\ell-3}2^{\ell+1} \text{ on some monotone path} \mid B \,]. \end{aligned}$$

This inequality follows from $\sum_{\ell\geq 3} m_{\ell-3}2^{\ell+1} \leq \frac{16}{n}\sum_{\ell=0}^{2\log n} r_\ell 2^\ell \leq 32 d_{\text{ave}} n = 64n$. Hence,

$$\begin{aligned} & \Pr\left[\, \bar{A} \mid B \,\right] \\ = & \sum_{\ell:\ell\geq 3, m_{\ell-3}\geq \log n} \Pr[\text{ There is a monotone path containing } \geq \beta m_{\ell-3} \text{ delays from level } \ell \,] \\ \leq & \sum_{\ell:\ell\geq 3, m_{\ell-3}\geq \log n} 2^{2m_{\ell-3}} \Pr[\text{ A fixed monotone path } P \text{ has } \geq \beta m_{\ell-3} \text{ delays from level } \ell \,]. \end{aligned}$$

The equality follows from the definition of event B. For each level ℓ, where $\ell \geq 3$ and $m_{\ell-3} \geq \log n$, we go through the argument of case 2 in Theorem 19. In particular, for level ℓ network H is divided into $m_{\ell-3}^2$ boxes of size $\frac{n}{m_{\ell-3}} \times \frac{n}{m_{\ell-3}}$, and Lemma 20 is applied to bound $E[e^{\lambda X^\ell}]$, where X^ℓ is the random variable that counts the number of level ℓ delays on path P. Therefore,

$$\begin{aligned} \Pr\left[\, \bar{A} \mid B \,\right] & \leq \sum_{\ell:\ell\geq 3, m_{\ell-3}\geq \log n} e^{-(\lambda\beta-\delta-2)m_{\ell-3}} \\ & \leq 2\log n e^{-(\lambda\beta-\delta-2)\log n}. \end{aligned}$$

When $\beta = 10$ this probability is less than n^{-1}. Since the delays from levels 0-2 contribute a delay of at most $16n$ on a monotone path, the proof is complete. □

Non-constant Average Delay

Suppose network H has a non-constant average delay d_{ave}. We build network $\bar{H}$ from H by "shrinking" the delays. If edge $\bar{e}$ in $\bar{H}$ corresponds to edge e in H, and $d_{\bar{e}}$ and d_e are delays on the two edges respectively, then $d_{\bar{e}}$ is equal to $\max\{d_e/(d_{\text{ave}}/2), 1\}$. One can verify that the average delay for $\bar{H}$ is at most 3. Hence, the length of the longest monotone path in $\bar{H}$ is $O(n)$ with high probability. It follows that the length of the longest monotone path in H is $O(d_{\text{ave}} n)$ with high probability.

Since we can always find an $m \times m$ subsquare of H of average at most d_{ave} to simulate G, Theorem 21 implies that the slowdown is $\max\{n^2/m^2, d_{\text{ave}} m/n\}$ with high probability. When $m = n/d_{\text{ave}}^{1/3}$, we have,

Theorem 22 *Suppose the delays on network H is a random permutation of a set of n^2 delays of average d_{ave}, then H can simulate G with slowdown $O(d_{\text{ave}}^{2/3})$ with high probability.*

Acknowledgements

The authors wish to thank Bobby Blumofe, Charles Leiserson, Bruce Maggs and Larry Rudolph for many helpful comments.

References

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

[2] M. Andrews, T. Leighton, P. T. Metaxas, and L. Zhang. Improved methods for hiding latency in networks of workstations. *Submitted to SPAA*, 1996.

[3] 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.

[4] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP, San Diego, CA*, pages 102–112. ACM Press, New York, NY, 1993.

[5] 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.

[6] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovoc, P. Raghavan, S. Roa, 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

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