Automatic Methods for Hiding Latency in Parallel and Distributed Computation

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. For example, given any "dataflow" type of algorithm that runs in T steps on an n-node ring with unit link delays, we show how to run the algorithm in $O(T)$ steps on any n-node bounded-degree connected network with *average* link delay $O(1)$. This is a significant improvement over prior approaches to latency hiding, which require slowdowns proportional to the *maximum* link delay. In the case 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 efficiently simulate an $n \times n$ array with unit link delays using slowdown $\tilde{O}(d_{\text{ave}}^{2/3})$ on a 2-dimensional array with average link delay d_{ave}. Lastly, we present results for the case in which large local databases are involved in the computation.

Keywords: Hiding latency, parallel and distributed computation, linear and 2-dimensional arrays, complementary slackness.

Subject classification: 68Q22 Parallel and distributed algorithms.

1 Introduction

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

*Bell Laboratories. Murray Hill, NJ. Supported by NSF contract 9302476-CCR and ARPA contract N00014-95-1-1246. Email: andrews@research.bell-labs.com. Work performed while at MIT.

†Department of Mathematics and Laboratory for Computer Science, MIT. Cambridge, MA. 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. Wellesley, MA. Supported by NSF contract 9504421-CCR and ARPA contract N00014-95-1-1246. Email: pmetaxas@wellesley.edu.

§Bell Laboratories. Murray Hill, NJ. Supported by an NSF graduate fellowship and ARPA contract N00014-95-1-1246. Email: ylz@research.bell-labs.com. Work performed while at MIT.

1.1 Traditional Approaches

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

An alternative approach is to organize the network in a hierarchical fashion so that the latencies are consistent with the hierarchy. For example, the CM-5 [1, 14] is organized into a fat tree and the KSR consists of two levels of nested rings. In both cases, the highest latency links are segregated into the top levels of the network hierarchy. This type of architecture works well for applications in which most of the computation is local since local computation can proceed using the low-level low-latency links. Only rarely, it is hoped, would the high latency links be needed. Thus, only certain steps of the computation would be slow. Unfortunately, this approach is not suitable for scenarios where the network is unstructured (which is often the case for a NOW) or when the underlying application requires frequent communications through the high-level links.

Redundant computation is another approach that has been used in the past [6, 11, 13] to hide the effects of latency. Here the idea is to avoid latency by recomputing data locally instead of waiting to receive it through a high-latency link.

Probably the most generally applicable method of hiding latency is the approach known as *complementary slackness*. The idea behind this approach is to load each processor with enough work so that it stays productive while waiting for data to be supplied by the network. There are many implementations and incarnations of this method. For example, each processor in the CRAY YMP C-90 keeps busy by operating on a pipeline of 128 64-bit words. Processors on the HEP machine [21] swapped between unrelated threads while waiting for the data. The CM-1 and CM-2 were designed to simulate much larger virtual machines so that a single processor would perform the computation of many virtual processors [4, 22]. The technique also forms a critical component of Valiant's bulk synchronous model of parallel computing [23, 24] and it has been employed in several algorithms papers [3, 10, 11, 15, 20].

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

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

1.2 Model and Problem

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

Dataflow Model

In the dataflow model, each computation solely depends on the computation of the previous step. Creating a pebble (i,t) involves two time units. The first time unit is for communication, where g_i obtains pebbles of the form $(j, t-1)$ from all its neighbors g_j. The second time unit is for computation, where g_i performs computation based on pebbles $(j, t-1)$ and records the result in pebble (i,t). Take an example of an n-node guest linear array. In $2T$ time steps, G creates $n \times T$ pebbles, where pebble (i,t), for $1 < i < n$ and $1 < t \leq T$, depends on pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$. (See Figure 1.) Any host processor p can compute pebble (i,t) as long as p has the information in pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$, either by directly computing these pebbles or by receiving them from neighboring processors.

The dataflow model is applicable to many computations such as matrix operations, Fourier transform, sorting, algorithms for computational geometry, etc. A large number of examples can be found in [12].

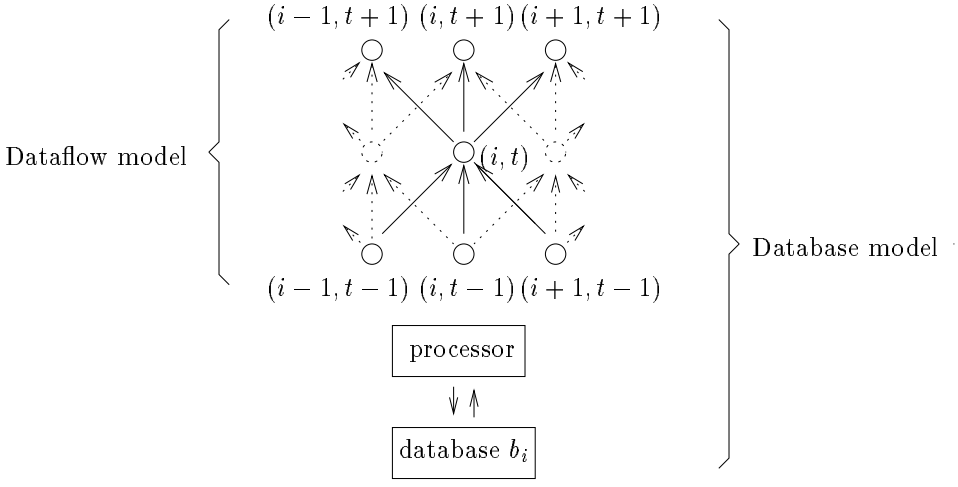


Figure 1: The computation pebbles created by a guest linear array.

Database Model

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

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

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

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

Bandwidth

The guest network G has unit bandwidth on each link. This allows each pebble to be passed along a unit-delay link of G in one time step. In our simulation we assume that the link bandwidth of the host network H is w. That is, P pebbles can be passed along a d-delay link of H in $d + \lceil \frac{P}{w} \rceil - 1$ steps by pipelining. In many cases of our study, it is sufficient to assume that the host and the guest have comparable link bandwidth, i.e. w is a constant. However, in certain situations the bandwidth needs to be $\tilde{O}(\log n)$. Otherwise, we pay an extra factor of $\tilde{O}(\log n)$ in the slowdown. The details are discussed in Sections 3.2.4 and 4.2.4.

1.3 Results

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

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

Table 1: Result Summary.

The first two results in Table 1 are proved in terms of linear arrays. An n-node unit-delay ring is essentially the same as an n-node unit-delay linear array, since the latter can simulate the former with a slowdown of 2 [12]. Result 1 is asymptotically optimal in some cases. In addition, we also have a constant-approximation algorithm for simulating rings and linear arrays in the dataflow model. Results 2 and 3 are optimal up to a polylogarithmic factor in some cases. Result 3 is for a worst-case model. When the delays on the host are randomly arranged, the bound can be improved to $O(d_{\text{ave}}^{2/3})$. Results 4 and 5 are easy generalizations of Results 1 and 2 respectively. Sections 2 and 3 present latency hiding methods for the dataflow model. Section 4 concentrates on the database model.

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

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

In Section 4, we also establish limits on the degree to which the high latency can be mitigated when each database is allowed a small number of copies. For example, if each database has only one copy, we show that the slowdown can be as much as $d_{\max}$ even if d_{ave} is a constant and the best simulation is used. When each database has at most two copies and each host processor copies a constant number of databases, we give an example of a host whose average delay is a constant, but for which the slowdown has a lower bound of $\Omega(\log n)$. These results demonstrate that it is easier to overcome latencies in dataflow types of computations than in computations that require access to large local databases.

1.4 A Related Scheduling Problem

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

the given host processors so as to minimize the *makespan*, i.e. the total time taken to execute all the tasks.

A variation of the above scheduling problem has been studied. Here, we are given any task dag (not necessarily created by a guest network in the dataflow model). All the arcs in the dag are associated with a fixed quantity that indicates the communication cost. Note that, unlike our problem, the communication cost here is the same for any processor-pair. In [18] Papadimitriou and Ullman studied an $n \times n$ grid dag (which they called a diamond dag). They showed a nontrivial time-communication tradeoff and gave an asymptotically-optimal schedule. Their result was similar to the special case of our Result 1 stated in Section 1.3 where all the link delays in our host network are the same. In [19] Papadimitriou and Yannakakis presented a 2-approximation algorithm for general dags where an unlimited number of processors could be used. For well-known families of dags such as the full binary tree, the diamond dag and the fast Fourier transform, only a finite number of processors were needed and their approximation algorithms were optimal (or near-optimal). Redundant computation was used in [19].

Dag scheduling has been studied in other papers, including [2, 5, 7, 8, 9, 16, 17]. Some variations of the problem include the cases in which the dags are limited to certain topologies, the task nodes require different execution times, arcs require different communication time and/or processors have different processing powers.

2 Dataflow Model – Linear Arrays

We begin our presentation with the methods for hiding latency in linear arrays. Our basic approach is to transfer a process that involves a two-way communication to a process that involves one-way communication only. (This idea is also essential for simulating 2-dimensional arrays in Section 3.) We present an asymptotically tight bound on the slowdown for linear arrays. All the results for linear arrays are applicable to rings.

2.1 Average Delay – An Upper Bound

Let the network G be an n-processor guest linear array with unit delay on all the edges. Let the network H be an n-processor host linear array with arbitrary delays, where d_i is the delay on the ith edge of H. As discussed in Section 1.2, in $2T$ time steps G creates $n \times T$ pebbles, where pebble (i,t), for $1 < i < n$ and $1 < t \leq T$, depends on pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$. We first present algorithm STRIPE in which H simulates G with a slowdown of $O(d_{\text{ave}})$, where $d_{\text{ave}} = \sum_{k=1}^{n-1} d_k/(n-1)$ is the average delay of H.

Consider the first $n/2$ rows of pebbles created by G. Let L be the triangle formed by pebbles (i,t), where $i+t \leq n+1$. Let R be the triangle formed by pebbles (i,t), where $i \leq t$. (See Figure 2.) In STRIPE, H first simulates the bottom half of L and then the bottom half of R. At this point every pebble in the first $n/2$ rows is simulated. If the entire computation of G is partitioned into groups each of which consists of $n/2$ rows of pebbles, then H can repeat the process and simulate every group in a similar manner.

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

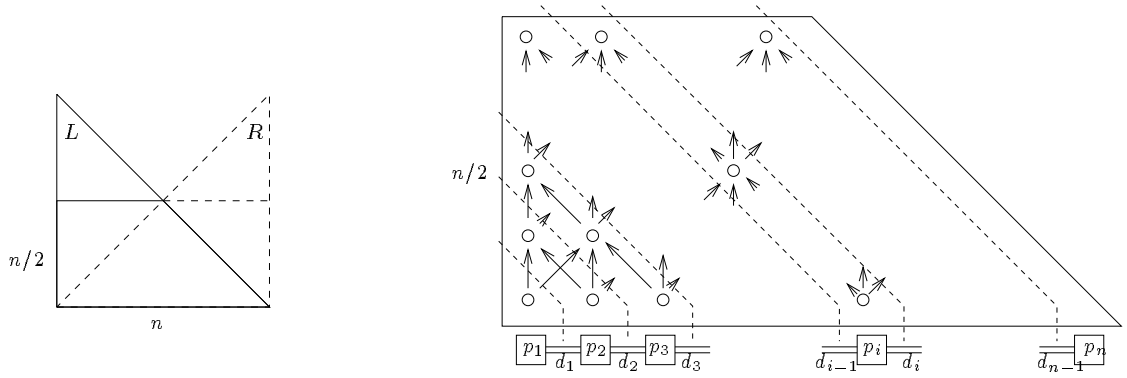


Figure 2: (Left) Triangles L and R. (Right) Algorithm STRIPE. Each slanted stripe is simulated by one processor of H. Arrows correspond to communications. Dashed lines correspond to the delays d_i encountered by communications.

Lemma 2.1 *Processor p_i $(1 \le i \le n)$ 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 $0 \le i \le n/2$, 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. (Note that the intersection of R and L only needs to be computed once.) Thus, H has completed simulating the first $n/2$ rows of pebbles created by G. To continue the simulation, each pebble $(i, n/2)$ is passed to processor p_i. With pipelining, this can be done in $\sum_{k=1}^{n-1} d_k$ steps. The next $n/2$ and every subsequent $n/2$ rows of pebbles can be simulated in a similar manner. Therefore, the slowdown is upper bounded by,

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

2.2 A Better Upper Bound

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

Lemma 2.2 *Processor p_i finishes simulating stripe i by step $\ell n/2 + \sum_{k=1}^{i-1} d_k$.*

Proof: We inductively show that p_i can compute the pebbles in the xth row of stripe i by time step $\ell x + \sum_{k=1}^{i-1} d_k$. The base of the induction holds trivially for $i = 1$ and $x = 1$, since processor p_1 does not depend on other processors and pebbles in the first row do not depend on other pebbles.

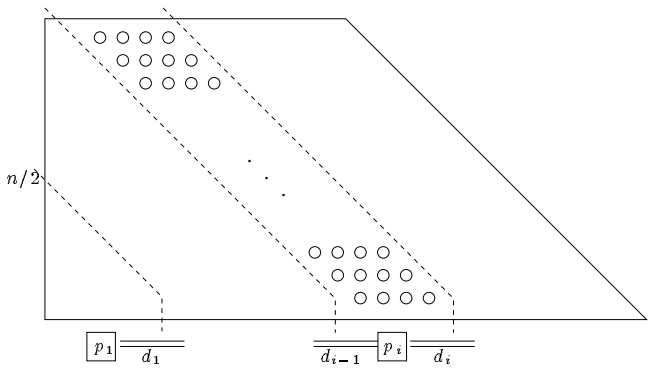


Figure 3: Algorithm FatStripe. Processor p_i simulates stripe i which has width $\ell = n/m$. (In the figure, $\ell = 4$.) All the pebbles in stripe i are computed by time step $\ell n/2 + \sum_{k=1}^{i-1} d_k$.

Let us consider the pebbles on the $(x+1)$st row of stripes $i+1$, for $x \geq 1$ and $i \geq 1$. These pebbles could only depend on pebbles on the xth row of stripe $i-1$, i and $i+1$, which can be computed by processors p_{i-2}, p_{i-1} and p_i by steps $\ell x + \sum_{k=1}^{i-3} d_k$, $\ell x + \sum_{k=1}^{i-2} d_k$ and $\ell x + \sum_{k=1}^{i-1} d_k$ respectively by induction. Hence, p_i is able to receive all the information necessary to compute its $(x+1)$st row by step $\ell x + \sum_{k=1}^{i-1} d_k$ and therefore finish computing the $(x+1)$st row by step $\ell(x+1) + \sum_{k=1}^{i-1} d_k$. Since each stripe contains at most $n/2$ rows, p_i finishes simulating stripe i by step $\ell n/2 + \sum_{k=1}^{i-1} d_k$. □

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

Theorem 2.3 FatStripe *achieves a slowdown of* $\min_{\text{intervals } I} O(n/m_I + d_I m_I/n)$.

In the case when $\sqrt{d_{\text{ave}}} \leq n$, there exists an interval I with $M_I = n/\sqrt{d_{\text{ave}}}$ processors and average delay $d_I \leq d_{\text{ave}}$ by the pigeon-hole principle. Theorem 2.3 implies that the slowdown is $O(\sqrt{d_{\text{ave}}})$ when M_I simulates G. In the case when $\sqrt{d_{\text{ave}}} > n$ a single host processor is used to carry out the simulation, which incurs a slowdown of $n = O(\sqrt{d_{\text{ave}}})$. The simulation is work-efficient in both cases. Therefore,

Corollary 2.4 FatStripe *efficiently simulates G on H and achieves a slowdown of $O(\sqrt{d_{\text{ave}}})$, where d_{ave} is the average delay of H.*

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

2.3 A Matching Lower Bound

We proceed to show that the upper bound, $\min_I O(n/m_I + d_I m_I/n)$, in Theorem 2.3 is asymptotically tight by showing that $\min_I \max\{n/2m_I, d_I m_I/2n\}$ is a lower bound on the best achievable slowdown even if we allow *redundant computation*. Note that with redundant computation, a pebble may be computed by several host processors. This technique makes it more likely for the host

to simulate the guest efficiently. However, we show below that redundancy does not help in this case.

Lemma 2.5 *The top pebble, $(1, n)$ of triangle L, cannot be computed at a time step earlier than*

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

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

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

It follows that the slowdown in simulating triangle L is lower bounded by τ/n. By a similar argument to Lemma 2.5 none of the pebbles (i, n), for $1 \leq i \leq n$, can be computed at a time step earlier than τ. By repeating this argument the first kn rows of G cannot be simulated in time less than $k\tau$. Therefore, we obtain,

Theorem 2.6 *The slowdown of any simulation of an n-node guest linear array G by a host linear array H is lower bounded by $\min_I \Theta(n/m_I + d_I m_I/n)$, where I is a subarray of H and has m_I processors and average delay d_I. Hence,* FatStripe *is optimal up to a constant factor.*

2.4 Simulating Linear Arrays on General Networks

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

Lemma 2.7 *Let H be a connected n-node network with arbitrary topology. Then an n-node linear array $\mathcal{H}$ can be one-to-one embedded in H such that every edge of H is used at most twice in $\mathcal{H}$.*

Proof: Our proof follows the approach of Theorem 3.15 in [12, page 470]. We include the proof here for completeness. It is sufficient to embed a linear array $\mathcal{H}$ in a spanning tree of H. The proof proceeds by induction on the height of the tree with the following inductive hypothesis. For any

child u of the root v, there is a one-to-one embedding of a linear array in the tree such that v and u form two endpoints of the array, the edge uv is used at most once and all other edges of the tree are used at most twice. (Note that we treat all the edges as undirected.)

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

We place the next node of the linear array at any child x of u_2 (if any), using edges u_1v, vu_2 and u_2x. Again, we inductively place nodes of the linear array in the subtree rooted at u_2 such that u_2 and x are endpoints. We continue in this fashion. At the last subtree rooted at u, we enter this subtree at a child of u (if any) and exit at u. This completes the embedding of the linear array. Our lemma follows from the observation that the linear array has endpoints v and u, edges vu_1, ..., vu_{d-1} are used twice and vu_d is used once. (See Figure 4.)

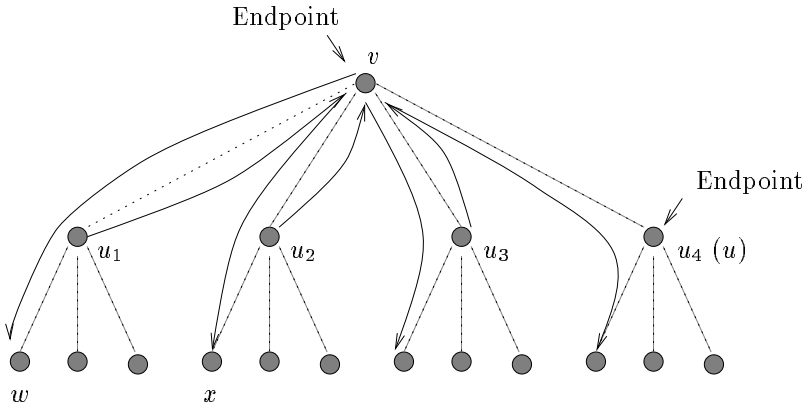


Figure 4: Embed a linear array one-to-one in a tree such that each tree edge is used at most twice. The dotted lines indicate tree edges and the solid lines indicate array edges.

□

Since H has n nodes and degree δ, H has at most $\delta n/2$ edges and therefore the total delays on all edges of H is at most $\delta d_{\text{ave}} n/2$. By Lemma 2.7, $\mathcal{H}$ uses each edge of H at most twice. Hence, the total delays on all edges of $\mathcal{H}$ is at most $\delta d_{\text{ave}} n$, and the average delay of $\mathcal{H}$ is at most δd_{ave}. By Corollary 2.4, $\mathcal{H}$ can simulate G with a slowdown of $O(\sqrt{\delta d_{\text{ave}}})$. When H has bounded degree, i.e. $\delta = O(1)$, we have,

Theorem 2.8 *A bounded-degree host network with average delay d_{ave} can efficiently simulate an n-processor guest linear array with a slowdown of $O(\sqrt{d_{\text{ave}}})$.*

Theorem 2.8 does not hold when H has unbounded degree. Consider the following example. Let H be a linear array of $\sqrt{n}$ cliques, in which each clique contains $\sqrt{n}$ nodes. If a clique edge has delay 1 and an edge connecting two adjacent cliques has delay n, then H has $d_{\text{ave}} < 4$. Suppose m connected cliques are used to simulate n steps of G. Lemma 2.5 implies a slowdown of $\min_m \max\{\sqrt{n}/2m, m/2\}$ in simulating every n steps of computation by the guest. The first term

follows from a work argument, since $m\sqrt{n}$ processors are in m cliques. The second term comes from the communication delay, since a linear array embedded in these m connected cliques has a total delay of at least mn. Hence, the slowdown is at least $\min_m \max\{\sqrt{n}/2m, m/2\}$, which is $\Omega(n^{1/4})$, whereas the average delay is a constant.

3 Dataflow Model – Two-Dimensional Arrays

In this section we present methods for hiding latency in 2-dimensional arrays. The analysis here is substantially more complex than that for the 1-dimensional case. We focus on simulating a 2-dimensional array on a 2-dimensional array. Section 3.1 generalizes the approach for the linear arrays. Section 3.2 introduces some new mechanism to improve the bound. Section 3.3 discusses the case when the delays are randomly arranged.

3.1 An Analogue of the One-Dimensional Case

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

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

Algorithm 2D-RAY is a 2-dimensional analogue of STRIPE. To simulate the first $n/2$ steps of computation of G, 2D-RAY simulates P_1, P_2, P_3 and P_4 one by one. Pyramid P_1 is divided into n^2 rays, each of which is simulated by one processor of H. In particular, processor $p_{i,j}$ of H simulates ray $R_{i,j}$, consisting of pebbles $(i-t+1, j-t+1, t)$, for $1 \leq t \leq \min\{i, j, n/2\}$. (See Figure 5.) When every pebble for the first $n/2$ steps of computation of G is simulated, 2D-RAY repeats the process and simulates the next $n/2$ steps of computation. In the following we bound the slowdown in terms of the total delay on *monotone paths*, where a monotone path travels in two directions, up and right. Let the length of a path be the total delay on the path, and let $D_{i,j}$ be the length of the longest monotone path from processor $p_{1,1}$ to $p_{i,j}$ in H. We have,

Lemma 3.1 *Processor $p_{i,j}$ of H is able to compute pebble $(i-t+1, j-t+1, t)$ at step $D_{i,j}+t$.*

Proof: We use induction on the indices (i,j) of the processors. The base of the induction for $p_{1,1}$ is obvious. Pebble $(i-t+1, j-t+1, t)$ depends on pebbles $(i-t+1, j-t+1, t-1)$, $(i-t, j-t+1, t-1)$, $(i-t+2, j-t+1, t-1)$, $(i-t+1, j-t, t-1)$ and $(i-t+1, j-t+2, t-1)$, which are computed by processors $p_{i-1,j-1}$, $p_{i-2,j-1}$, $p_{i,j-1}$, $p_{i-1,j-2}$ and $p_{i-1,j}$ respectively. (See Figure 5.) 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 $\max\{D_{i-1,j}+x_{i-1,j}, D_{i,j-1}+y_{i,j-1}\}+t = D_{i,j}+t$.

□

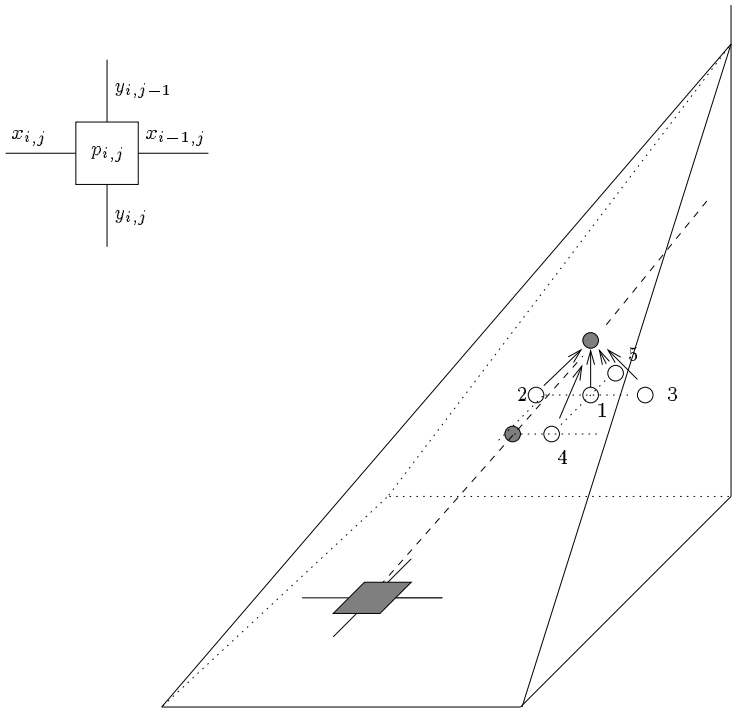


Figure 5: Algorithm 2D-RAY. In pyramid P_1 the dashed line represents the ray of pebbles computed by processor $p_{i,j}$ (which is shown in the upper left corner). Two of those pebbles, computed at times t and $t-1$, are shown shaded. The five numbered pebbles are those that $(i-t+1, j-t+1, t)$ depends on.

Hence, 2D-RAY simulates pyramid P_1 in $D_{n,n}+n$ steps. Since P_2, P_3, and P_4 can be simulated similarly, 2D-RAY simulate the first $n/2$ steps of computation of G in $O(D_{n,n}+n)$ steps. The simulation is repeated for every $n/2$ steps of computation of G. Therefore,

Lemma 3.2 *Algorithm* 2D-RAY *achieves a slowdown of* $O(D_{n,n}/n)$*, where* $D_{n,n}$ *is the length of the longest monotone path in* H*.*

Unfortunately, $D_{n,n}$ can be large compared with d_{ave}, the average delay of H. In the worst case $D_{n,n}$ can be $\Theta(n^2 d_{\text{ave}})$, implying a slowdown of $\Theta(n d_{\text{ave}})$. We introduce algorithm FATRAY, a two-dimensional analogue of FATSTRIPE, to achieve a slowdown that is often better than $O(D_{n,n}/n)$. Pyramid P_1 is divided into m^2 rays, each of which has size $\ell \times \ell = \frac{n}{m} \times \frac{n}{m}$. FATRAY uses an $m \times m$ contiguous subarray of processors in H to carry out the simulation. For simplicity, assume FATRAY uses processors $p_{i,j}$ $(1 \le i,j \le m)$. Again, $p_{i,j}$ computes every pebble in ray $R_{i,j}$, and $p_{i,j}$ first computes all the pebbles on the bottom plane and then moves up. The follow lemma is analogous to Lemma 2.2.

Lemma 3.3 *Processor* $p_{i,j}$ *finishes simulating ray* $R_{i,j}$ *by step* $\ell^2 n/2 + D_{i,j}$*.*

Proof: As in Lemma 2.2 we can inductively show that $p_{i,j}$ can compute all the pebbles in the xth plane in ray $R_{i,j}$ by time step $\ell^2 x + D_{i,j}$. Since each ray contains at most $n/2$ planes of pebbles, $p_{i,j}$ finishes simulating ray $R_{i,j}$ by step $\ell^2 n/2 + D_{i,j}$. □

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

Theorem 3.4 FATRAY *achieves a slowdown of* $\min_{\text{subarrays } S} O(n^2/m_S^2 + D_S/n)$.

Unfortunately, the slowdown can still be big compared with d_{ave}. For example, suppose that H is partitioned into n squares of size $\sqrt{n} \times \sqrt{n}$ with one edge of delay n in the center of each square and unit delay on all other edges. The slowdown is $\min_S \Theta(n^2/m_S^2 + D_S/n) = \Theta(n^{1/3})$ whereas d_{ave} is a constant. Matters are better, however, when all the delays are the same, as we show in the following theorem.

Theorem 3.5 *In the case where all the delays in H are d,* FATRAY *efficiently simulates G on H and achieves a slowdown of $\Theta\left(\min\{d^{2/3}, n^2\}\right)$. The slowdown is optimal up to a constant factor.*

Proof: When $d \leq n^3$ FATRAY uses a subarray of size $\frac{n}{d^{1/3}} \times \frac{n}{d^{1/3}}$. Theorem 3.4 implies a slowdown of $O(d^{2/3})$. We show that the slowdown is asymptotically tight as follows. Consider pebble $(i, j, d^{1/3})$, and suppose processor q computes it in a simulation. Let A be the set the pebbles of the form (i', j', t), for $1 \leq t < d^{1/3}$, on which $(i, j, d^{1/3})$ depends, i.e. $(i, j, d^{1/3})$ cannot be computed until after (i', j', t) is computed. If every pebble in A is computed by q then it takes at least $|A| = \Omega\left((d^{1/3})^3\right) = \Omega(d)$ time steps to simulate A. Otherwise, a processor $p \neq q$ computes some pebble in A and passes this information to q. The delay from p to q is at least d. Hence, the slowdown on simulating the first $d^{1/3}$ steps is $d^{2/3}$. The same argument applies for the slowdown in the next $d^{1/3}$ steps.

When $d > n^3$ FATRAY uses a single host processor for the simulation and achieves a slowdown of $O(n^2)$. This slowdown is asymptotically tight for the same reason as in the previous case. We consider pebbles (i, j, n) instead of $(i, j, d^{1/3})$. In both cases the simulation is work-efficient. □

Theorem 3.5 can be generalized to any k-dimensional array, for $k \geq 1$.

Theorem 3.6 *Suppose G is an $n \times \ldots \times n$ k-dimensional array with unit-delay edges, and H is an $n \times \ldots \times n$ k-dimensional array with delay-d edges, then H can efficiently simulate G with a slowdown of $\Theta(\min\{d^{k/k+1}, n^k\})$. The slowdown is optimal up to a constant factor.*

3.2 Improved Bounds for Worst-Case Delays

In order to improve the slowdown, we observe that not all the host processors are useful. If a host processor is surrounded by high delays, then the benefit to be gained by using its computing power is nullified by the communication cost. We first describe criteria of removing such host processors. We then embed guest processors to the unremoved host processors. Suppose that guest processor $g_{i,j}$ is mapped to host processor p, then p computes the pebbles in ray $R_{i,j}$ in the 2D-RAY algorithm. For any arrangement of the delays in H, we show how to embed G on H such that, for any monotone path in G, its image in H has length of $O(d_{\text{ave}} n \log^{5/2} n)$. As a result, Lemma 3.2 implies a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$ as long as only $O(1)$ guest processors are mapped to each host processor. By applying the idea used in FATRAY, we improve the slowdown to $O(d_{\text{ave}}^{2/3} \log^{5/3} n)$ and achieve work-efficiency at the same time.

3.2.1 Removing Useless Processors

We first recursively represent H using a quad-tree, in which each node corresponds to a subarray of H. The root represents the entire $n \times n$ array. The four children of the root represent the four $\frac{n}{2} \times \frac{n}{2}$ subarrays, etc. In general, a node at depth k of the quad-tree corresponds to an $\frac{n}{2^k} \times \frac{n}{2^k}$ subarray of H. We refer to this subarray as a *depth-k* array. The leaves represent the individual processors of H. (See Figure 6.)

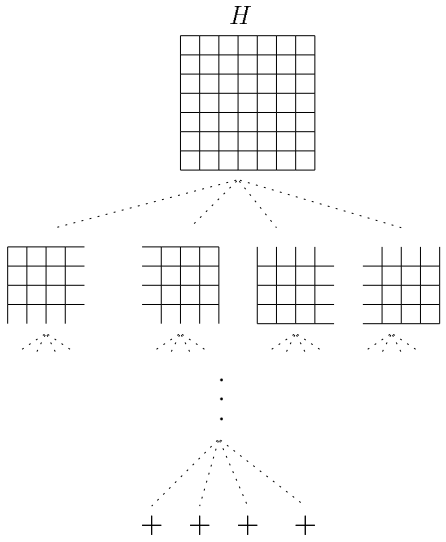


Figure 6: The quad-tree that represents H.

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

$$D_k = (c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right) \tag{1}$$

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

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

$$k_{\max} = \log n - \frac{1}{2}\log c - \frac{1}{2}\log\log n. \tag{3}$$

- **Stage 1** From depth $k = k_{\max}$ down to depth 0, if the total delay on a row/column of a depth-k array exceeds the threshold D_k, then all the $\frac{n}{2^k}$ processors on that row/column are removed.
- **Stage 2** From depth $k = k_{\max}$ down to depth 0, if the number of unremoved processors in a depth-k array is smaller than the threshold m_k, then all the processors in that array are removed. Moreover, we also remove processors so that the number of remaining processors in any depth-k array is an integer multiple of m_k.

Lemma 3.7 *At most $2n^2/c$ processors are removed in stage 1.*

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

Lemma 3.8 *At most n^2/c processors are removed at stage 2.*

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

We label each array with the number of unremoved processors contained in it. By Lemmas 3.7 and 3.8, at most $3n^2/c$ processors of H are removed. Therefore, H is labeled with $c_1 n^2$, where $c_1 \geq 1 - (3/c)$. Any constant $c > 3$ works for our argument.

3.2.2 The Embedding

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

EMBED partitions $\mathcal{G}$ into *regions* recursively, and each depth-k region of $\mathcal{G}$ corresponds to a depth-k array of H. The depth-0 region is the entire network $\mathcal{G}$. By the construction of stage 2, $c_1 n^2$ (the number of processors in $\mathcal{G}$) is a multiple of m_0. Hence, $\mathcal{G}$ can be viewed as a collection of contiguous squares of size $\sqrt{m_0} \times \sqrt{m_0}$. We inductively assume that each depth-k region consists of contiguous squares of size $\sqrt{m_k} \times \sqrt{m_k}$, where m_k is defined in Equation (2). Each depth-k region R is partitioned into four depth $k+1$ regions R_1, R_2, R_3 and R_4 as follows. First, each $\sqrt{m_k} \times \sqrt{m_k}$ square of R is divided into four squares of size $\sqrt{m_{k+1}} \times \sqrt{m_{k+1}}$, where $\sqrt{m_{k+1}} = \sqrt{m_k}/2$. Suppose that R_i corresponds to a depth $k+1$ square of H that has z_i unremoved processors, then R_i has size z_i. By the construction of stage 2, z_i is a multiple of m_{k+1}. Hence, R_i can be formed as a collection of contiguous squares of size $\sqrt{m_{k+1}} \times \sqrt{m_{k+1}}$. Note that if z_i is 0, then the corresponding R_i is empty. (See Figure 7.)

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

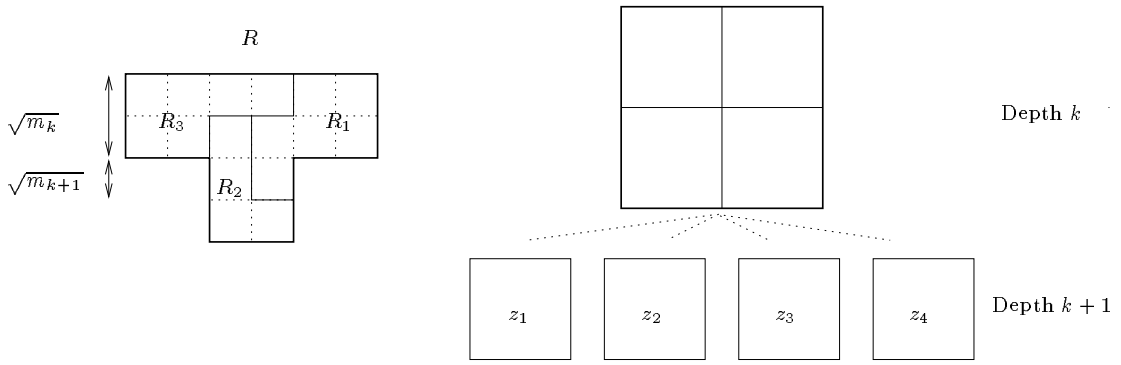


Figure 7: (Left) Depth-k region R and depth $k+1$ regions R_1, R_2 and R_3 of $\mathcal{G}$. (Right) Depth-k and $k+1$ arrays of H. Depth $k+1$ region R_i has size z_i, where z_i is the number of unremoved processors in the corresponding array of H. In this figure, $z_4 = 0$, and R_4 is therefore empty.

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

3.2.3 Bounding Monotone Path Length

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

Lemma 3.9 *Let P be any monotone path in $\mathcal{G}$. The image of P in H has a total delay of $O(d_{\text{ave}} n \log^{5/2} n)$ under* EMBED.

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

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

Hence, we can embed G on H such that $O(1)$ guest processors are mapped to each host processor and that the image in H of any monotone path in G has length $O(d_{\text{ave}} n \log^{5/2} n)$. Lemma 3.2 implies that H can simulate G with a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$. To improve the slowdown and achieve efficiency, we apply the idea of complementary slackness and use an $m \times m$ contiguous subarray of H for simulation as in FATRAY. Theorem 3.4 and Lemma 3.9 imply a slowdown of $O\left((d_{\text{ave}} m \log^{5/2} m)/n + n^2/m^2\right)$. By choosing m to be $\max\left\{n d_{\text{ave}}^{-1/3} \log^{-5/6} n, 1\right\}$, we have,

Theorem 3.10 *Network H with average delay d_{ave} can efficiently simulate G with a slowdown of $O\left(d_{\text{ave}}^{2/3} \log^{5/3} n\right)$.*

3.2.4 Bandwidth

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

In Section 3.2.3, peers a_H and b_H are connected by a 1-bend route in H. To address the congestion issue, we present a more sophisticated method of connecting a_H and b_H such that each edge in H has $O(\log^{3/2} n)$ routes going through it and that the distance between a_H and b_H remains unchanged asymptotically.

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

Lemma 3.11 *Consider any depth-k array, S_k, of H. The number of processors in S_k that have peers outside S_k is $O(x\sqrt{\log n})$, where x is the number of live rows and columns in S_k.*

Proof: Let z be the number of unremoved processors in S_k, then the number of live rows and columns is at least $\frac{z}{n/2^k}$. The number of host processors in S_k that have peers outside S_k is proportional to the perimeter of R_k, the depth-k region that corresponds to S_k. By the construction of EMBED, R_k consists of squares of size $\sqrt{m_k} \times \sqrt{m_k}$. Hence, R_k has perimeter of $O(z/\sqrt{m_k})$, which is $O\left(\frac{z}{n/2^k}\sqrt{\log n}\right)$ by the definition of m_k in Equation (2). Our lemma follows. □

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

Fact 3.12 *Consider a routing problem on a square array of size $x \times x$.*

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

Our recursive routing starts at depth $k = k_{\max}$. Consider all the depth-k arrays S_k. For all the peers that are k-related, we connect them through a 1-bend routing within S_k. Since S_k has size $\sqrt{\log n} \times \sqrt{\log n}$ and each host processor has at most 4 peers, the congestion caused by this 1-bend routing within S_k is $O(\sqrt{\log n})$ by item 1 of Fact 3.12. For all the processors that have peers outside S_k, we route them to live boundary points such that the following two conditions hold. First, each live boundary point of S_k receives $O(\sqrt{\log n})$ requests. This is possible because of Lemma 3.11. Second, the routing uses 1 bend and causes a congestion of $O(\log n)$ by item 1 of Fact 3.12.

We proceed recursively to depths $k < k_{\max}$. Consider all the depth-k arrays S_k. From the previous stage the host processors that are not connected to their peers are routed to some live boundary points of depth $k+1$ arrays. Hence, they are either on the boundary or on the cross of S_k, and $O(\sqrt{\log n})$ host processors are routed to the same location. For all the peers that are k-related, we connect them within S_k. Otherwise, we route them to the live boundary points of S_k such that each live point receives $O(\sqrt{\log n})$ requests (including those from all previous stages but have not yet connected to their peers). This is possible by Lemma 3.11. In both cases, item 2 of Fact 3.12 implies that the routing can be done in $O(1)$ bends and that the congestion incurred is $O(\sqrt{\log n})$.

The congestion incurred at depth k, for $1 \le k < k_{\max}$, is $O(\sqrt{\log n})$ and at depth $k_{\max}$ is $O(\log n)$. Since each of the depths uses the same underlying edges, the overall congestion is $O(\log^{3/2} n)$. The host processors are routed to live boundary points in $O(1)$ bends at each depth, and therefore the length incurred at depth k is $O\left(\frac{n}{2^k} d_{\text{ave}} \log n\right)$. Suppose that a_H and b_H are k-related, then the distance between them is $\sum_{k' \ge k} O\left(\frac{n}{2^{k'}} d_{\text{ave}} \log n\right)$, which remains $O\left(\frac{n}{2^k} d_{\text{ave}} \log n\right)$ as in Lemma 3.9. In summary,

Lemma 3.13 *In the above routing scheme the congestion is $O(\log^{3/2} n)$ on all edges of H. Furthermore, for any monotone path P in $\mathcal{G}$, the image of P in H has length $O(d_{\text{ave}} n \log^{5/2} n)$.*

3.3 Improved Bounds for Randomly-Arranged Delays

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

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

3.3.1 Shortcuts and Edge Coloring

If an edge with a large delay is surrounded by edges with small delays, we can route around this large delay. The intuition is that in a random permutation most long delays can be shortcut. For each edge f, we consider four edge-disjoint *alternate* paths that connect the two endpoints of f. (See Figure 8.) The 3×3 box that contains these four paths is called the *bounding box* of f. After the *shortcut*, the total delay on f equals the shortest alternate path length. For clarity, we shall refer to the delay before the shortcut as the *original delay* and the delay after the shortcut as the *shortcut delay*.

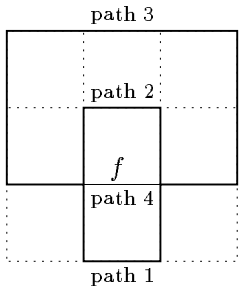


Figure 8: The bounding box and four edge-disjoint alternate paths for edge f.

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

Lemma 3.14 *The probability for an edge f to have a level-ℓ^+ shortcut delay is $O(2^{-4\ell})$.*

Proof: If edge f has a shortcut delay from level ℓ^+, then the four alternate paths must each have an edge whose original delay is from level $(\ell - 4)^+$. For a particular set of four edges to have level $(\ell - 4)^+$ original delays, the probability is $\binom{B}{4}/\binom{M}{4}$, where B is the number of level $(\ell - 4)^+$ original delays, and M is the number of edges in H. Since there are $3 \cdot 3 \cdot 9 \cdot 1 = 81$ ways to choose four edges from four alternate paths, we derive the following from a union bound.

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

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

Unfortunately, these probabilities are *not* independent from edge to edge for two reasons. First, the arrangement of delays is a permutation of a given set of delays. This does not cause a problem however, as the analysis in Lemmas 3.15 and 3.17 will show. Intuitively, in a permutation if one edge has a large delay then other edges are less likely to have large delays. Second, the bounding boxes are not necessarily disjoint. To resolve this problem we introduce an *edge coloring*, so that any two distinct edges with the same color have edge-disjoint bounding boxes. Clearly, only a constant number of colors are needed.

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

3.3.2 Large Delays

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

Lemma 3.15 *With probability* $1 - O(n^{-1})$, any *monotone paths pick up a total delay of* $O(n)$ *from levels* $\ell \geq L$, *where* $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.

Proof: By Lemma 3.14, the probability that one particular edge has a shortcut delay from level $(\frac{3}{4}\log n)^+$ is $O(n^{-3})$. Since H has $\Theta(n^2)$ edges, with probability $1 - O(n^{-1})$ no edge in H has shortcut delay from level $(\frac{3}{4}\log n)^+$.

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

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

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

We bound probability p with the inequalities,

$$\left(\frac{y}{x}\right)^x \leq \binom{y}{x} \leq \left(\frac{ye}{x}\right)^x, \tag{5}$$

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

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

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

3.3.3 Small Delays

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

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

Lemma 3.16 *With probability $1 - O(n^{-2})$, any monotone path picks up a total of $O(n2^{-\ell})$ delay from level-ℓ shortcut delays, where $\ell < L$ is one particular level and $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.*

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

$$\begin{aligned} P &= \Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,] \\ &\leq 81^k \cdot \binom{A}{k_1} \cdot \ldots \cdot \binom{A}{k_{2m-1}} \cdot \binom{B}{4k} \Big/ \binom{M}{4k}. \end{aligned}$$

By Inequality (5), the probability is bounded by,

$$\begin{aligned} P &\leq 81^k \cdot \left(\frac{Ae}{k_1}\right)^{k_1} \cdot \ldots \cdot \left(\frac{Ae}{k_{2m-1}}\right)^{k_{2m-1}} \cdot \left(\frac{Be}{M}\right)^{4k} \\ &= \prod_{i=1}^{2m-1} \left(\frac{81 \cdot a \cdot b^4 \cdot e^5}{2^4 \cdot k_i}\right)^{k_i}. \end{aligned} \tag{6}$$

We proceed to bound the expectation of e^X.

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

The first inequality follows from Inequality (6), and the last equality follows from $e^{y(2m-1)} = \left(\sum_{j\geq 0} \frac{y^j}{j!}\right)^{2m-1}$. We use Markov's inequality to bound the probability that the total number of level-ℓ shortcut delays exceeds $\beta(2m-1)$ in this particular monotone sequence.

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

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

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

Lemma 3.17 *With probability $1 - O(n^{-1})$, all the monotone paths pick up a total delay of $O(n)$ from levels $\ell < L$ for $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.*

For the case in which $d_{\text{ave}} = O(1)$, Lemmas 3.15 and 3.17 show that any monotone path has a total delay of $O(n)$ with high probability. For the case in which d_{ave} is nonconstant, the discussion at the beginning of section implies that,

Theorem 3.18 *Suppose H is a network with average delay d_{ave}, then with high probability every monotone path in H has delay $O(nd_{\text{ave}})$.*

To make the algorithm work-efficient, we use an $m \times m$ subarray of H of average delay at most d_{ave} to simulate G. Theorems 3.4 and 3.18 implies that the slowdown is $O(n^2/m^2 + d_{\text{ave}}m/n)$ with high probability. By choosing m to be $\max\{nd_{\text{ave}}^{-1/3}, 1\}$, we have,

Theorem 3.19 *Suppose the delays on network H are from a random permutation of a set of delays whose average is d_{ave}, then with high probability H can simulate G with slowdown $O(d_{\text{ave}}^{2/3})$.*

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

4 Database Model

We switch our attention to the database model. As discussed in Section 1, simulation in the database model is more difficult than in the dataflow model. For algorithms such as STRIPE in Section 2 to work for the database model a host processor needs $\Theta(n)$ copies of the databases on average. This is unrealistic because of the memory requirement as well as the difficulty in updating the databases. We therefore develop new machinery for the database model. Contrary to the dataflow model, we make substantial use of redundant computation. Apart from the slowdown, another important parameter for the database model is *load*, which is the number of databases that a host processor copies.

The main contribution of this section is an algorithm called OVERLAP that simulates linear arrays in the database model with a small load and a small slowdown. Since OVERLAP is technically involved, we begin with a special case in Section 4.1 where the host linear array has delay d on all edges. The simulation in this special case is much simpler, and it conveys some intuition for using redundant computation in the general case. Section 4.2 presents OVERLAP in detail. The techniques are generalized to simulate linear and 2-dimensional arrays on general networks in Sections 4.3 and 4.4. Lastly in Section 4.5 we discuss the lower bounds on slowdown when each database is allowed a small number of copies.

4.1 A Special Case

In this section we consider a special case. Let G be a guest linear array with n processors and unit-delay edges, and let H be a host linear array with n processors and delay d on all edges. We use redundant computation to achieve an optimal slowdown of $O(\sqrt{d})$. Recall that in the dataflow model, the optimal slowdown is achieved for linear arrays without using redandancy.

Theorem 4.1 *In the database model, H can efficiently simulate G with a slowdown and a load of $O(\sqrt{d})$. This slowdown is optimal up to a constant factor.*

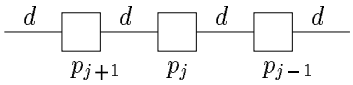
Proof: We consider two cases. If $n \leq \sqrt{d}$ then one host processor copies all the databases and carries out the entire computation by itself. Hence the load and the slowdown are n, which is $O(\sqrt{d})$. Otherwise, the first $\frac{n}{\sqrt{d}}$ host processors are used for the simulation. For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

We show how to simulate the first $\sqrt{d}$ rows of pebbles created by G in $O(d)$ steps by H. Every subsequent $\sqrt{d}$ rows of pebbles are simulated in the same manner. The algorithm is demonstrated in Figure 9. For $1 \leq j \leq \frac{n}{\sqrt{d}}$ let,

$$
\begin{aligned}
P_j &= \{\text{Pebbles } (i,t): && 1 \leq t \leq \sqrt{d}, && -2\sqrt{d}+1 \leq i - j\sqrt{d} \leq \sqrt{d}\},\\
L_j &= \{\text{Pebbles } (i,t): && 1 \leq t \leq \sqrt{d}, && 1 \leq i - (j-2)\sqrt{d} \leq t\},\\
R_j &= \{\text{Pebbles } (i,t): && 1 \leq t \leq \sqrt{d}, && -t+1 \leq i - (j+1)\sqrt{d} \leq 0\},\\
T_j &= P_j - (L_j \cup R_j),\\
A_j &= \{\text{Pebbles } ((j-2)\sqrt{d},t): && 1 \leq t \leq \sqrt{d}\},\\
B_j &= \{\text{Pebbles } ((j-1)\sqrt{d}+1,t): && 1 \leq t \leq \sqrt{d}\},\\
C_j &= \{\text{Pebbles } ((j\sqrt{d},t): && 1 \leq t \leq \sqrt{d}\},\\
D_j &= \{\text{Pebbles } ((j+1)\sqrt{d}+1,t): && 1 \leq t \leq \sqrt{d}\}.
\end{aligned}
$$

Processor p_j of H computes all the pebbles in P_j. First, p_j computes the pebbles in the trapezium T_j without communicating with its neighbors. There are $2d$ pebbles in T_j and so this takes $2d$ steps. Next, p_j passes column B_j to processor p_{j-1} and receives column A_j from p_{j-1}. It also passes column C_j to processor p_{j+1} and receives column D_j from p_{j+1}. This communication takes $d + \sqrt{d} < 2d$ steps using pipelining. Processor p_j can now compute the pebbles in triangles L_j and R_j in d steps. It is important for p_j to compute the pebbles in L_j and R_j in order to continue the simulation of the next $\sqrt{d}$ rows of pebbles, since databases need to be updated. This presents a major difference between the dataflow and database models.

Hence, it takes at most $5d$ steps in total for processor p_j to compute every pebble in P_j. The next $\sqrt{d}$ steps of computation can be simulated in a similar fashion. The slowdown is therefore $O(\sqrt{d})$. The lower bound proof in Theorem 3.5 implies that the slowdown of $\Omega(\sqrt{d})$ is necessary.



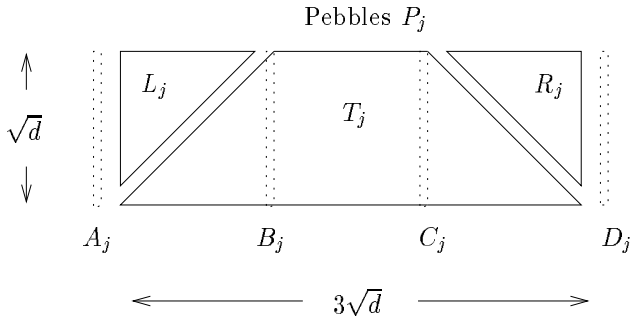


Figure 9: Simulating $\sqrt{d}$ steps of computation of G on H.

Note that during the computation of T_j, the pebbles in columns B_j and C_j can start to travel to the neighboring processors of p_j as soon as they are ready. Processor p_j can also start to compute triangles L_j and R_j before the entire columns of A_j and D_j are transfered. In this way, the communication time can be saved. Although it does not make a difference asymptotically in this case we take advantage of this observation in OVERLAP. □

4.2 Algorithm OVERLAP

To simulate a guest linear array on a host linear array with arbitrary delays we use an algorithm called OVERLAP. In OVERLAP, we remove host processors that are surrounded by high delays. The motivation of this step is similar to that of Section 3.2. For the remaining processors, we decide how much redundancy is needed for neighboring processors and how much computation each processor is able to carry out. During the simulation, some pebbles are redundantly computed to ensure that the communication is not too costly. We first obtain a slowdown of $O(d_{\text{ave}} \log^3 n)$, where d_{ave} is the average delay of H and n is the size of G and H, and later improve the slowdown to $O(\sqrt{d_{\text{ave}}} \log^3 n)$ while achieving work efficiency.

4.2.1 Removing Useless Processors

We recursively represent H using a binary tree, in which each node corresponds to a subarray of H. The root represents the entire array. The left and right children of the root represent the left and right halves of the array respectively. In general, a node at depth k of the binary tree corresponds to a subarray of H that contains $\frac{n}{2^k}$ processors. We refer to this subarray as a *depth-k interval*. The leaves represent the individual processors of H. (See Figure 10.)

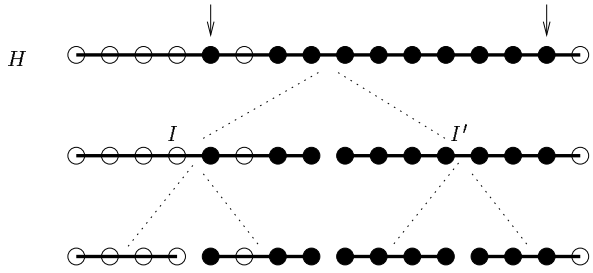


Figure 10: The binary tree that represents H. In this figure, unremoved processors of H are represented by black circles; removed processors are represented by white circles. Arrows indicate the endpoints of the root interval. Interval I has one live child and I' has two live children.

We describe a 2-stage process that removes the processors that are surrounded by high delays (stage 1) and the processors that are surrounded by few unremoved processors (stage 2). During stage 2, we also label each *live* subarray, where a live subarray contains some unremoved processor. These labels indicate how many columns of pebbles the live subarrays are able to compute.

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

$$D_k = (c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right) \tag{7}$$

$$m_k = \left(\frac{1}{c \log n}\right)\left(\frac{n}{2^k}\right). \tag{8}$$

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

$$k_{\max} = \log n - \log\log n - \log c. \tag{9}$$

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

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

  Suppose all the live depth-$(k+1)$ intervals are labeled. Now consider each live depth-k interval I. If I has two live children I_1 and I_2 that are labeled with x_1 and x_2, then let $x = x_1 + x_2 - m_{k+1}$. If I has one live child I_1 that is labeled with x_1, then let $x = x_1$. If

$x < 2m_k$, we remove all the remaining processors in I and I is no longer live. Otherwise, we label I with x. We proceed to depth $k-1$ until reaching depth 0.

Lemma 4.2 *At most n/c processors are removed at stage 1.*

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

Lemma 4.3 *The label on the root interval is at least $\left(1 - \frac{5}{2c}\right) n$ at stage 2.*

Proof: Before stage 2, the number of remaining processors in H is at least $(1-1/c)n$ by Lemma 4.2. At depth $k = k_{\max}$ of stage 2, the sum of the labels on the live depth-k intervals is at least $(1-1/c)n - 2m_k 2^k$, which is $(1-1/c)n - \frac{2n}{c \log n}$. At each depth $k < k_{\max}$, the sum of the labels on the live depth-k intervals decreases by at most $(2m_k + m_{k+1})2^k$, which is $\frac{5n}{2c \log n}$. Summing over all depths, we conclude that the label at the root interval is at least $\left(1 - \frac{5}{2c}\right) n$. □

4.2.2 Assigning Databases

For clarity of presentation, we first assume that G has n' processors, where n' is the label on the root interval of G and n' is a constant fraction of n by Lemma 4.3. We also assume the existence of pebbles $(0,t)$ and $(n'+1,t)$, for all $t \geq 1$, which are known to H at time step 0. This ensures that each pebble computed by G is dependent on three pebbles.

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

4.2.3 The Simulation

In OVERLAP, H recursively simulates every $m_0 = \frac{n}{c \log n}$ rows of pebbles created by G as follows. If H, the depth-0 interval, has two live depth-1 intervals I_1 and I_2 as children, then I_1 and I_2 recursively compute the first $m_1 = m_0/2$ rows of pebbles and then repeat for the next m_1 rows. In particular, I_1 (resp. I_2) computes all the pebbles of the form (i,t), where I_1 (resp. I_2) owns database b_i and $1 \leq t \leq m_1$. Intervals I_1 and I_2 share m_1 databases and therefore redundantly compute these m_1 columns of pebbles. If H has one live child I_1, then I_1 recursively computes the first m_1 rows and then repeats for the second m_1 rows. At depth $k = k_{\max}$, each depth-k interval computes $m_k = 1$ row of pebbles. Theorem 4.4 explains the simulation in details.

Let us define a set of values $s_t^{(k)}$ for $0 \leq k \leq k_{\max}$ and $1 \leq t \leq m_k$, where the superscript k represents the depth of the recursion, and the subscript t represents the row number. Roughly speaking, $s_t^{(k)}$ corresponds to the time by which a depth-k interval computes its pebbles in the tth

row. We are interested in the slowdown $s_{m_0}^{(0)}/m_0$, where $s_{m_0}^{(0)}$ corresponds to the time that H takes to simulate the first m_0 steps of computation by G. Recall that the delay threshold D_k defined in Equation (7) is an upper bound on the total delay in any live depth-k interval. The recurrences are as follows.

$$s_t^{(k)} = s_t^{(k+1)} + D_k \qquad \text{for} \qquad 1 \le t \le m_{k+1} \tag{10}$$

$$s_t^{(k)} = s_{t-m_{k+1}}^{(k)} + s_{m_{k+1}}^{(k)} \qquad \text{for} \quad m_{k+1}+1 \le t \le m_k \tag{11}$$

The base of the recurrence is defined to be,

$$s_{m_k}^{(k)} = s_1^{(k)} = 1 \qquad \text{for} \quad k = k_{\max}. \tag{12}$$

Let the *left endpoint* of interval I be the leftmost unremoved processor in I, and the *right endpoint* be the rightmost unremoved processor in I. (See Figure 10.) For notational simplicity, we assume that I is the leftmost live depth-k interval and is assigned databases $b_1, \ldots, b_x$. Let $B_k = \{(i,t) : 1 \le i \le x, 1 \le t \le m_k\}$. The proof of the following theorem describes how algorithm OVERLAP performs the simulation.

Theorem 4.4 *For $1 \le t \le m_k$, if pebbles $(0,t)$ and $(x+1,t)$ are known by time step $s_t^{(k)}$ by the left and right endpoints of interval I respectively, then by time step $s_t^{(k)}$ every pebble (i,t) in B_k is computed by all the processors in interval I that have a copy of database b_i.*

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

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

Case 1: Suppose I has two live children I_1 and I_2 that are labeled with x_1 and x_2 respectively. By construction $x = x_1 + x_2 - m_{k+1}$. Let $B_{k+1} = \{(i,t) : 1 \le i \le x_1,\ 1 \le t \le m_{k+1}\}$. Let $y = x_1 - m_{k+1}$ and $B'_{k+1} = \{(i,t) : y+1 \le i \le x,\ 1 \le t \le m_{k+1}\}$. Let column C consist of pebbles (y,t) and column D consist of pebbles (x_1+1,t), where $1 \le t \le m_{k+1}$. Note that boxes B_{k+1} and B'_{k+1} have an overlap of width m_{k+1}, i.e. the m_{k+1} columns between C and D are common to both B_{k+1} and B'_{k+1}. (See Figure 11.) Two observations can be made from the inductive hypothesis.

- **Observation 1** For $1 \le t \le m_{k+1}$, every pebble (y,t) in column C can be computed by I_1 by time step $s_t^{(k+1)}$ *without any conditions on pebbles* $(0,t)$ *and* (x_1+1,t). Since C and D are m_{k+1} columns apart and $x_1 \ge 2m_{k+1}$ by the construction of stage 2, the pebbles in column C therefore do not depend on the pebbles $(0,t)$ and (x_1+1,t). (The dotted diagonal lines in Figure 11 show the dependencies of columns C and D.)

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

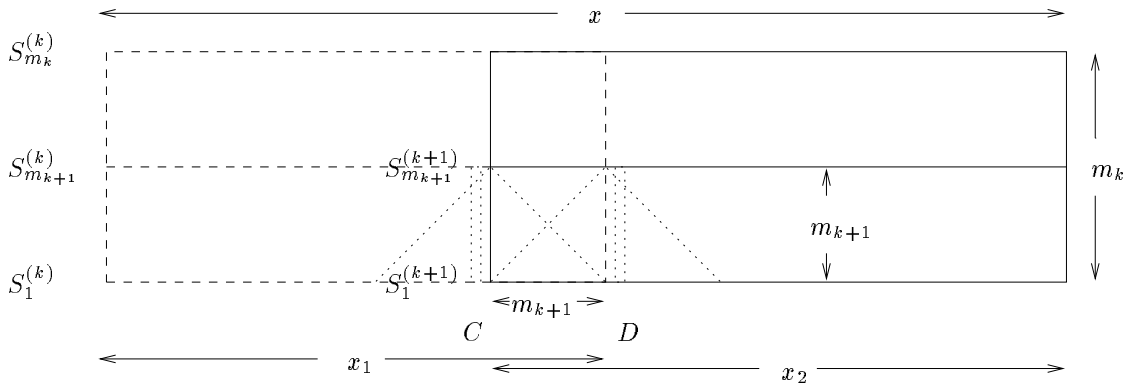


Figure 11: The box of pebbles B_{k+1} has size $x_1 \times m_{k+1}$ and is represented by the lower left box with a dashed boundary. B'_{k+1} has size $x_2 \times m_{k+1}$ and is represented by the lower right box with a solid boundary. B_k is the union of all four boxes. For interval I to compute every pebble in B_k, I_1 and I_2 (the live children of I) recursively compute B_{k+1} and B'_{k+1}. Once the bottom half of B_k is computed the top half is computed in a similar manner.

Similar statements can be made about the box B'_{k+1} and column D. Now suppose that the value of pebbles $(0,t)$ and $(x+1,t)$ are known at time step $s_t^{(k)}$ by the left and right endpoints of interval I respectively. Observation 1 and the inductive hypothesis imply that any pebble (y,t) in column C can be computed by I_1 by time $s_t^{(k+1)}$. Since the total delay in interval I is at most D_k then the left endpoint of interval I_2 can receive the pebble (y,t) (together with any relevant database changes) by time $s_t^{(k+1)} + D_k$ which equals $s_t^{(k)}$ (Equation (10)). Similarly, all of the pebbles in column D can be sent to the right endpoint of interval I_1 by time $s_t^{(k)}$. Since $s_t^{(k)}$ is greater than $s_t^{(k+1)}$ by a constant amount, namely D_k, for $1 \leq t \leq m_{k+1}$, Observation 2 and the inductive hypothesis imply that pebbles (i,t) in box B_{k+1} (resp. B'_{k+1}) are computed by I_1 (resp. I_2) by time $s_t^{(k)}$. Therefore, pebbles (i,t) in the bottom half of B_k are computed by time $s_t^{(k)}$.

Once the bottom half of B_k is simulated I simulates the top half in a similar manner. Thus, pebbles (i,t) in the top half of B_k are computed by time $s^{(k)}_{m_{k+1}} + s^{(k)}_{t-m_{k+1}}$ which equals $s_t^{(k)}$ (Equation (11)).

Case 2: The case in which I has one live child is simpler. Let I_1 be the child of I. By construction, I_1 has label $x_1 = x$. By Observation 2 and the induction hypothesis, if the values of the pebbles $(0,t)$ and (x_1+1,t), for $1 \leq t \leq m_{k+1}$, are known at time steps $s_t^{(k)}$ by the left and right endpoints of interval I_1 respectively, then every pebble (i,t) in B_{k+1} (i.e. the bottom half of B_k) is computed by I_1 by time step $s_t^{(k)}$. Since intervals I and I_1 have the same remaining processors (and hence the same endpoints), the above statement holds for I. Interval I then computes the top half in the same manner. Thus, pebbles (i,t) in the top half of B_k are computed by time $s^{(k)}_{m_{k+1}} + s^{(k)}_{t-m_{k+1}}$ which equals $s_t^{(k)}$ (Equation (11)).

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

Recall that n' is the label of the tree root and n' is a constant fraction of n by Lemma 4.3. We have the following.

Theorem 4.5 *Suppose that guest linear array G has n' processors and the host linear array H has n processors and an average delay of d_{ave}. Algorithm* OVERLAP *simulates G with H such that the load on H is one and the slowdown is $O(d_{\text{ave}} \log^3 n)$.*

Proof: The load on H follows directly from the database assignment. The box B_0 contains all of the pebbles for the first m_0 steps of computations by G, where $m_0 = \frac{n}{c \log n}$. The root interval I_0 contains all the remaining processors of H. Since pebbles $(0, t)$ and $(n' + 1, t)$ are available at time step 0 by assumption, Theorem 4.4 implies that I_0, i.e. H, computes the pebbles in box B_0 by time $s^{(0)}_{m_0}$. We derive $s^{(0)}_{m_0}$ from the recurrence of $s^{(k)}_t$ in Equations (10) and (11) and the definition of D_k in Equation (7).

$$s^{(0)}_{m_0} = 2^k s^{(k)}_{m_k} + 2kD_0 \qquad \text{for } k = k_{\max}. \tag{13}$$

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

4.2.4 Bandwidth

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

4.2.5 Improvements

In this section we first modify OVERLAP to achieve work efficiency. So far each host processor is assigned at most 1 database, and the base of the recurrence is therefore $s^{(k)}_{m_k} = 1$ for $k = k_{\max}$ as defined in Equation (12). Observe that in Equation (13) the second term of $s^{(0)}_{m_0}$ dominates the first term. We can balance the two terms by increasing the value of $s^{(k)}_{m_k}$ for the base case, i.e. increasing the load on the host processors.

In particular, we use an m-processor subarray of the host linear array H to simulate an n-processor guest linear array, where $m = \max\left\{1, \frac{n}{d_{\text{ave}}} \log^{-3} n\right\}$ and the subarray has average delay at most d_{ave}. If $m = 1$, the slowdown and the load are both n. Otherwise, we carry out the 2-stage process to remove the useless processors of the m-processor subarray as as described in Section 4.2.1. The only difference is that the network size is m instead of n, and the variables such as D_k, m_k and $k_{\max}$ are also defined in terms of m. Each unremoved host processor is assigned $\Theta\left(\frac{n}{m}\right)$ databases, and hence the base case is $s^{(k)}_{m_k} = \Theta\left(\frac{n}{m}\right)$ for $k = k_{\max}$. We obtain,

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

from Equation (13). Since $m = \frac{n}{d_{\text{ave}}} \log^{-3} n$, we have $s^{(0)}_{m_0} = O\left(n \log^{-1} \frac{n}{d_{\text{ave}}}\right)$. Since $m_0 = \frac{m}{c \log m}$, the slowdown $s^{(0)}_{m_0}/m_0$ is $O(d_{\text{ave}} \log^3 n)$. This implies that the simulation is work preserving.

Theorem 4.6 *In the database model, an n-processor guest linear array can be efficiently simulated by an n-processor host linear array with a slowdown and a load of $O\left(d_{\text{ave}} \log^3 n\right)$, where the host has average delay d_{ave}.*

Combining Theorems 4.1 and 4.6 we can improve the slowdown by a factor of $O(\sqrt{d_{\text{ave}}})$ while preserving efficiency. Suppose that G is an n-processor guest linear array, and H is an n-processor host linear array with average delay d_{ave}. We make use of an intermediate linear array H_0 that has a delay of d_{ave} on every edge. Theorem 4.1 implies that network H_0 can efficiently simulate G with a slowdown of $O(\sqrt{d_{\text{ave}}})$, where $\max\left\{\frac{n}{\sqrt{d_{\text{ave}}}}, 1\right\}$ processors of H_0 are used. In the simulation by H_0, every $O(d_{\text{ave}})$ steps of computation interleave with every $O(d_{\text{ave}})$ steps of communication. If we treat every $O(d_{\text{ave}})$ steps as one time unit, then H_0 acts like a guest linear array with unit-delay edges and H has a normalized average delay of $O(1)$. Theorem 4.6 implies that H can simulate H_0 with a slowdown of $O(\log^3 n)$. The combined slowdown is therefore $O(\sqrt{d_{\text{ave}}}\log^3 n)$. It is obvious that the combined load is $O(\sqrt{d_{\text{ave}}}\log^3 n)$. Theorem 4.6 is improved to the following.

Theorem 4.7 *In the database model, an n-processor guest linear array can be efficiently simulated by an n-processor host linear array with a slowdown and a load of $O(\sqrt{d_{\text{ave}}}\log^3 n)$, where the host has average delay d_{ave}.*

4.3 Simulating Linear Arrays on General Networks

We generalize algorithm OVERLAP to simulate a guest linear array on an arbitrary bounded-degree connected host network. Given a connected bounded-degree n-processor network H with average delay d_{ave}, we first find a linear array $\mathcal{H}$ that can be embedded one-to-one to H and has average delay d_{ave}. As discussed in Section 2.4 such $\mathcal{H}$ can be found, and $\mathcal{H}$ is used to carry out the simulation. Combined with Theorem 4.7, we obtain,

Theorem 4.8 *An n-processor guest linear array can be efficiently simulated by a connected bounded-degree n-processor host with a slowdown of $O(\sqrt{d_{\text{ave}}}\log^3 n)$, where the host has average delay d_{ave}.*

For the same reason as in Section 2.4 Theorem 4.8 does not hold when H has unbounded degree.

4.4 Simulating 2-Dimensional Arrays on General Networks

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

Theorem 4.9 *In the database model, an $n \times n$ guest can be efficiently simulated by a bounded-degree host network with a slowdown of $O(n\log^3 n + \sqrt{nd_{\text{ave}}}\log^3 n)$, where the host has average delay d_{ave}.*

Proof: As discussed in section 2.4 there exists a linear array $\mathcal{H}$ such that $\mathcal{H}$ is embedded one-to-one in H and that $\mathcal{H}$ has average delay $O(d_{\text{ave}})$. The simulation of G on H will be performed by simulating G on $\mathcal{H}$. We first show how to simulate G on an intermediate linear array $\mathcal{H}_0$, where $\mathcal{H}_0$ has delay d_{ave} on all the edges. The size of $\mathcal{H}_0$ depends on the relative sizes of d_{ave} and n.

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

Since $d_{\text{ave}} < n$, if every $O(n)$ steps are treated as one time unit then $\mathcal{H}$ has a normalized average delay $O(1)$ and $\mathcal{H}_0$ acts like a guest linear array with unit-delay edges. Therefore, Theorem 4.7

implies that $\mathcal{H}$ can efficiently simulate $\mathcal{H}_0$ with a slowdown of $O(\log^3 n)$. The combined slowdown is therefore $O(n \log^3 n)$.

Case 2: If $d_{\text{ave}} \geq n$, then $\mathcal{H}_0$ has n/x processors, where $x = \sqrt{d_{\text{ave}}/n}$. Each processor of $\mathcal{H}_0$ simulates $3x$ columns of G, overlapping x columns with each neighbor. (The redundant computation used here is similar to that in Theorem 4.1.) To simulate x steps of G, each processor of $\mathcal{H}_0$ computes at most $3x^2 n$ pebbles and then communicates with both of its neighbors. The communication takes at most $3x^2 n + d_{\text{ave}}$ steps, which is $O(d_{\text{ave}})$ steps. Hence the slowdown of simulating every x steps is d_{ave}/x, which is $O(\sqrt{n d_{\text{ave}}})$. Also, in this simulation every $O(d_{\text{ave}})$ steps of computation interleave with every $O(d_{\text{ave}})$ steps of communication.

If every $O(d_{\text{ave}})$ steps is treated as one time unit, $\mathcal{H}$ has normalized average delay $O(1)$ and $\mathcal{H}_0$ acts like a linear array with unit-delay edges. If n/x processors of $\mathcal{H}$ are used to simulate $\mathcal{H}_0$, Theorem 4.7 implies a slowdown of $O(\log^3 \frac{n}{x})$, which is $O(\log^3 n)$. The combined slowdown is therefore $O(\sqrt{n d_{\text{ave}}} \log^3 n)$. □

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

Theorem 4.10 *In the dataflow model, an $n \times n$ guest can be efficiently simulated by a bounded-degree host network with a slowdown of $O(n + \sqrt{n d_{\text{ave}}})$, where the host has average delay d_{ave}.*

4.5 Lower Bounds

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

For the case in which each database is allowed one copy we give an example to show that the slowdown can be $d_{\max}$. Let G and H_1 be n-processor guest and host linear arrays. Every $\sqrt{n}$-th edge of H_1 has a delay of $\sqrt{n}$ and all other edges have unit delay. Therefore, H_1 has an average delay of $O(1)$. If at most $\sqrt{n}$ processors of H_1 have copies of databases, then by a work argument the slowdown when H_1 simulates G is at least $\sqrt{n}$. Otherwise, there exist databases b_i and b_{i+1} such that they are assigned to processors p and q of H_1 respectively and that the delay between p and q is at least $\sqrt{n}$. Hence, for all time steps t, processor p cannot compute pebble (i, t) until $\sqrt{n}$ steps after q computes $(i+1, t-1)$, and q cannot compute $(i+1, t)$ until $\sqrt{n}$ steps after p computes $(i, t-1)$. This implies a slowdown of $d_{\max} = \sqrt{n}$, whereas d_{ave} is a constant. Note that the above argument makes no assumption on the load.

Theorem 4.11 *If each database can have at most one copy, then there exists a host with $d_{\text{ave}} = O(1)$ such that the slowdown is $\Omega(\sqrt{n})$.*

For the case in which each database is allowed at most two copies we construct a host network H_2 whose average delay is $O(1)$, but for which the simulation slowdown is $\Omega(\log n)$. Network H_2 is made up of $\Theta(n)$ processors and the edge delays are either 1 or d. The following is a recursive construction of H_2 in which we define a series of boxes. (See Figure 12.) We regard H_2 as a level-k box, where $k = \log \frac{n}{d}$. Network H_2 consists of two level $k-1$ boxes that are connected by $\frac{2^k d}{\log n}$ edges of delay 1. In general, a level-ℓ box, for $1 \leq \ell \leq k$, consists of two level $\ell - 1$ boxes that are connected by $\frac{2^\ell d}{\log n}$ edges of delay 1. We say that these $\frac{2^\ell d}{\log n}$ processors are in a *segment*. A level-0 box consists of a single edge of delay d.

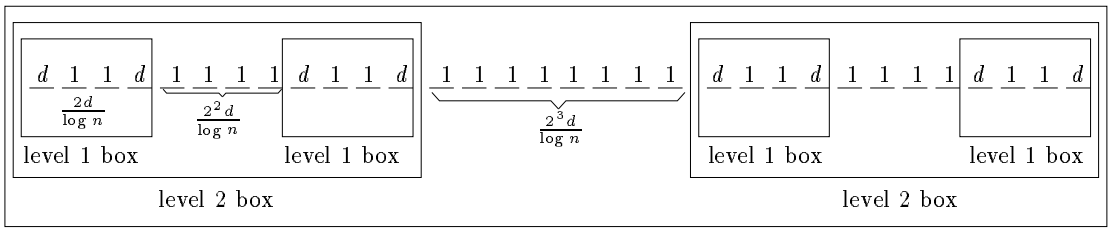


Figure 12: A level-3 box. Host network H_2 is a level-k box, where $k = \log \frac{n}{d}$.

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

Lemma 4.12 *If processors p and q are in two different segments I and J, then the delay between p and q is at least $\min\{\frac{u}{2}\log n, \frac{v}{2}\log n\}$, where u and v are the numbers of processors in segments I and J respectively. In particular, the delay between p and q is at least $d = \log n$.*

Theorem 4.13 *If each database is allowed at most two copies and the load is a constant c, then there exists a host with $d_{\text{ave}} = O(1)$ such that the slowdown is $\Omega(\log n)$.*

Proof: We consider the following two cases when H_2 simulates G.

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

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

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

This path goes backwards in time and zigzags during time steps k, for $k \in B \cup C \cup E \cup F$. (See Figure 13.)

By assumption processors p_k, for $k \in C \cup E$, can only belong to segment I or J. If processors p_k, for $k \in C \cup E$, do not belong to the same segment, then Lemma 4.12 implies a delay of $\frac{j}{2c}\log n$ for the communication between segments I and J. Hence, it takes more than $\frac{j}{2c}\log n$ steps to

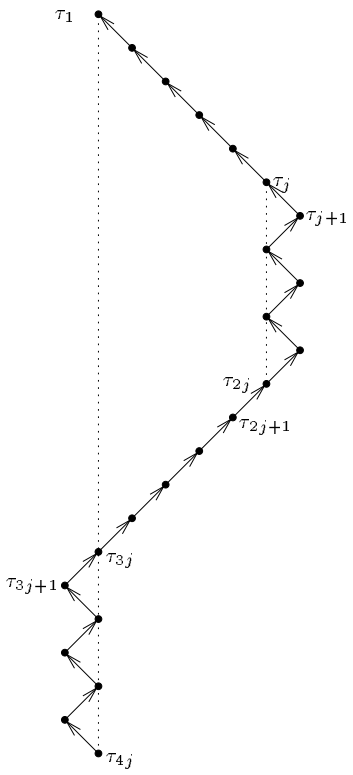


Figure 13: A path of $4j$ pebbles, where j is even.

compute this path of $4j$ pebbles. Otherwise, processors p_k, for $k \in C \cup E$, all belong to segment I. Lemma 4.12 implies a delay of $\log n$ in computing every τ_k, for $j < k \leq 2j$. This is because processors p_k, for $k \in B$, cannot be in segment I by assumption. Similarly, if processors p_k, for $k \in C \cup E$, all belong to segment J, then there is a delay of $\log n$ in computing every τ_k, for $3j < k \leq 4j$. Hence, it takes more than $j \log n$ steps to compute this path of $4j$ pebbles.

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

Case 2: There exists no "overlapping" of the databases as in case 1. Let $b_i, \ldots, b_j$, for $j \geq i$, be the longest sequence of consecutive databases assigned to one segment. Call this segment I and the sequence of databases S_I. Notice that processors in I do not have a copy of b_{i-1}. Let J be a segment that is assigned a copy of b_{i-1}. Let S_J be the sequence of consecutive databases such that b_{i-1} is a member of S_J and that each member of S_J has a copy in J. If b_i were a member of S_J, then either the database sequences S_J and S_I would produce the "overlapping" pattern sufficient for case 1 or S_J would be longer than S_I. This latter case contradicts with the definition of S_I. Hence, any segment that has a copy of b_{i-1} cannot have a copy of b_i. This implies that the processors computing the pebbles in the $(i-1)$st and ith column are at least $\log n$ delay apart by Lemma 4.12. Therefore, the slowdown is $\Omega(\log n)$. □

5 Conclusions

In this paper we presented methods for latency hiding in simple networks such as linear arrays and 2-dimensional arrays. Ultimately, we are interested in the efficient implementation of algorithms designed for networks that appear often in the architectures of parallel computers, such as trees,

arrays, butterflies and hypercubes, on a network with arbitrary topology and arbitrary link delays, such as NOWs. The special case in which two networks have identical topology but different link delays is a starting point where we can study the effect of latencies in isolation. Indeed, the general case of simulating a unit-delay guest on a host with arbitrary delays and arbitrary topology so as to minimize slowdown seems a very challenging problem.

References

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

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

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

[6] 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.

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

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

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

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

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

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

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