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

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

Abstract

In this paper we describe methods for mitigating the degradation in performance caused by high latencies in parallel and distributed networks. Our approach is similar in spirit to the "complementary slackness" method of latency hiding, but has the advantage that the slackness does not need to be provided by the programmer, and that large slowdowns are not needed in order to hide the latency. Our approach is also similar in spirit to the latency hiding methods of [2], but is not restricted to memoryless dataflow types of programs.

Most of our analysis is centered on the simulation of unit-delay rings on networks of workstations (NOWs) with arbitrary delays on the links. For example, given any collection of operations (including updates of large local memories or databases) that runs in t steps on a ring of n workstations with unit link delays, we show how to perform the same collection of operations in $O(t\log^3 n)$ steps on any connected, bounded-degree network of $n/\log^3 n$ workstations for which the *average* link delay is constant. (Here we assume that the bandwidth available on the NOW links is $O(\log n)$ times the bandwidth available on the ring links. An extra factor of $\log n$ appears in the slowdown without this assumption.) The result makes non-trivial use of redundant computation, which is required to avoid a slowdown that is proportional to the *maximum* link delay. The increase in memory and computational load on each workstation needed for the redundant computation is at most $O(1)$. In the case where the average latency in the network of workstations (d_{ave}) is not constant, then the slowdown needed for the simulation degrades by an additional factor of $O(\sqrt{d_{\text{ave}}})$. This is still far superior to a slowdown of $\Theta(d_{\max})$ which can occur without redundant computation.

As a consequence of our work on rings, we can also derive emulations of a wide variety of other unit-delay network architectures on a NOW with high-latency links. For example, we show how to emulate an N-node 2-dimensional array with unit delays, using slowdown $s = O(\sqrt{N}\log^3 N + N^{1/4}\log^3 N\sqrt{d_{\text{ave}}})$ on any connected bounded-degree network of $O(N/s)$ workstations with *average* link delay d_{ave}. The emulation is work-preserving and the slowdown is close to optimal for many configurations of the network of workstations.

We also prove lower bounds that establish limits on the degree to which the high latency links can be mitigated. These bounds demonstrate that it is easier to overcome latencies in dataflow types of computations than in computations that require access to large local databases.

1 Introduction

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

[*]Department of Mathematics and Laboratory for Computer Science, MIT. Supported by NSF contract 9302476-CCR, ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246. Email: andrews@math.mit.edu

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

[‡]Department of Computer Science, Wellesley College. Supported by NSF contract 9504421-CCR, ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246. Email: pmetaxas@wellesley.edu.

[§]Department of Mathematics and Laboratory for Computer Science, MIT. Supported by an NSF graduate fellowship, ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246. Email: ylz@math.mit.edu.

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, in the CM-5 [10, 1] the highest latency links are segregated into the top levels of the network hierarchy. This type of architecture works well for applications in which most of the computation is local since local computation can proceed using the low-level low-latency links. Only rarely, it is hoped, would the high latency links be needed. Thus, only certain steps of the computation would be slow. Unfortunately, this approach is not suitable for scenarios where the network is unstructured (which is often the case for a NOW) or when the underlying application requires frequent communications through the high-level links.

Redundant computation is another approach that has been used in the past [9, 5, 7] 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 [13] swapped between unrelated threads while waiting for the data. The CM-1 and CM-2 were designed to emulate much larger virtual machines so that a single processor would perform the computation of many virtual processors [14, 4]. The technique also forms a critical component of Valiant's bulk synchronous model of parallel computing [15, 16] and it has been employed in several algorithms papers [11, 7, 3, 12, 6].

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

Our goal in this paper 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 communication network, thereby easing the task of writing code. Moreover, our methods will enable us to automatically convert a program that was written for a well-structured unit-delay machine into a program that will run with minimal degradation in performance on a network with potentially large and variable latencies, at least for certain classes of networks.

In a previous paper [2], we devised automatic methods for hiding latency in a *dataflow* model of computation. In the dataflow model, the computation performed by a processor p at step t depends only on the results of the computations performed by p and its neighbors at the preceding step. This model is sufficiently general so as to be applicable for many algorithms (such as matrix operations, Fourier transforms, etc.) but it does not include applications where the computation performed by a processor depends on the state of a potentially large local memory or database. Nor does it contemplate a situation where part of the computation performed by a processor is to update its local memory. These limitations could be critical in some applications involving a network of workstations.

In this paper we devise automatic methods for hiding latency for a more general model of computation in which each processor p has a potentially large local memory that may be accessed and updated by p during each step. We refer to the local memory of each processor as the *database* for the processor, and to the model of computation as the *database model*. We assume that the initial contents of each database can be copied *before* the computation begins (thereby allowing replicated 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. It is possible to pass copies of *updates* performed on a database through the network, however. In general, we assume that a wire with delay d can transmit $P \log n$ "packets" of data, each containing an update, in $d + P - 1$ steps.[1]

Simulation in a database model is more difficult than in a dataflow model. Intuitively, this is because computation in a dataflow model is processor independent, and hence can be done by any processor with the information of the previous computation. The database model is more restricted, where 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 have each processor copy all the databases. This is because memory is expensive and it is difficult to keep every copy of the databases updated.

We begin by devising methods for hiding latency in a ring architecture. In particular, we show how to simulate a guest ring G with unit-delay links on an n-processor host ring H with arbitrary delays on the links. (Actually, our results are described in terms of linear arrays, but since a linear array can simulate a ring with slowdown 2 [8], the distinction is not important.) *A*

[1]Our algorithm assumes that the bandwidth available on the host links is $\log n$ times larger than the bandwidth on the guest links. This assumption can be removed if we pay an extra factor of $\log n$ in the slowdown.

priori, it would seem that any such simulation would require slowdown $d_{\max}$, where $d_{\max}$ is the largest delay in H. (Indeed, this is the delay that would be incurred by most prior approaches, although it should be pointed out that the prior approaches could preserve efficiency by using only $n/d_{\max}$ of the processors of H to carry out the simulation.) The result in this paper not only preserves efficiency but also accomplishes a slowdown of $O(\sqrt{d_{\text{ave}}}\log^3 n)$, where H is an n-processor host ring with an average delay of d_{ave} and G is a $(\sqrt{d_{\text{ave}}}n\log^3 n)$-processor guest ring. The slowdown is particularly impressive when $d_{\max} \gg \sqrt{d_{\text{ave}}}\log^3 n$, which is often the case in NOWs. More generally, a similar result holds whenever H is an arbitrary connected bounded-degree fixed-connection network. Our simulation also achieves a minimum *load* (up to a constant factor), i.e. each host processor only has copies of $O(\sqrt{d_{\text{ave}}}\log^3 n)$ databases.

Our method makes substantial use of redundant computation. In fact, we prove that redundant computation is necessary to hide latency for generic computations in the database model. This represents a substantial difference from the work in [2] for the dataflow model, for which redundant computation is apparently not useful in hiding latency.

As a consequence of our work on rings, we can also derive efficient emulations of a wide variety of other unit-delay network architectures on a NOW with high-latency links. For example, we show how to emulate an N-node 2-dimensional array with unit delays, using slowdown $s = O(\sqrt{N}\log^3 N + N^{1/4}\log^3 N\sqrt{d_{\text{ave}}})$ on any connected bounded-degree network of $O(N/s)$ workstations with average link delay d_{ave}. The emulation is work-preserving and the slowdown is close to optimal for many configurations of the NOW.

We also prove lower bounds that establish limits on the degree to which the high latency links can be mitigated. When 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. When each database has at most two copies and the host processors have a constant load, 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 bounds demonstrate that it is easier to overcome latencies in dataflow types of computations than in computations that require access to large local databases.

The remainder of the paper is divided into sections as follows. In section 2 we give a detailed description of the database model. In section 3, we describe latency hiding methods for ring computations. These results are generalized to other kinds of computations and NOWs in sections 4 and 5. In section 6 we discuss the lower bounds on slowdown when each database is allowed to have at most two copies. We conclude with several open questions in section 7.

2 The Model

We consider the problem of simulating an m-processor linear array G with unit-delay links on an n-processor linear array H with arbitrary delays on its links. We refer to G as the *guest* and H as the *host*. Let $g_1, \ldots, g_m$ be the processors of G. Each processor g_i owns a database b_i. Databases are consulted before each computation and updated after each computation. *Pebble* (i,t) represents the computation by processor g_i at time step t. In particular, at time t, processor g_i consults its database b_i and performs a computation based on b_i and pebbles $(i-1,t-1)$, $(i,t-1)$ and $(i+1,t-1)$. (See Figure 1.) The computation is done in one time step and the result

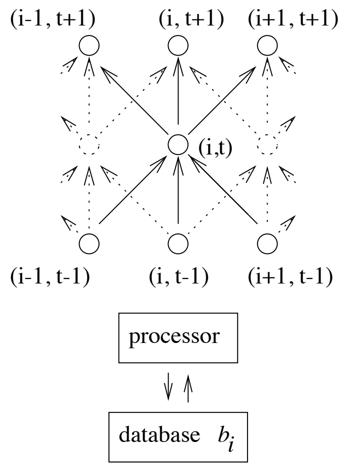


Figure 1: The computation of pebbles.

is recorded in pebble (i,t). Processor g_i then updates b_i.

In a *simulation* of G, H performs the same step-by-step computations as G when G is used in a general purpose way, i.e. H computes every pebble created by G. In the simulation, a pebble not only records the result of a computation but also the changes to the database incurred by this computation. Notice that a pebble does not contain a snapshot of the whole database but only the changes incurred by one computation. A pebble therefore has small size. We assume that the bandwidth available on the links of the host network H is $\log n$ times larger than the bandwidth on the links of the guest network G. (This assumption can be removed by paying an extra factor of $\log n$ in the slowdown of our simulations.) Hence, P pebbles can be passed along a d-delay link in $d + \lceil \frac{P}{\log n} \rceil - 1$ steps. Databases can be arbitrarily big and thus cannot be passed along the links.

Before the simulation starts, processors $p_1, \ldots, p_n$ of H decide which databases to copy. Suppose processor p of H copies databases b_i and b_j. Since databases are too big to be passed along the links, p only has access to databases b_i and b_j and hence p can only compute pebbles in columns i and j (i.e. pebbles of the form (i,t) and (j,t) for $t \geq 1$) during the simulation. Moreover, if both processors p and q decide to copy b_i, then p and q each has a copy of b_i, and p and q can only look up and update its own copy. We say that processor p *knows* a pebble if p either computes this pebble or receives it from some other processor. If p is to compute pebble (i,t) then p must know pebbles $(i-1,t-1)$, $(i,t-1)$ and $(i+1,t-1)$ and have an updated copy of b_i. Since databases cannot be passed along the links, p must also know all the pebbles (i,t'), for $1 \leq t' < t$, in order to update the changes in b_i incurred by these operations.

The *load* is defined to be the maximum number of databases that a processor of H copies. An algorithm achieves *minimum load* if the load is $O(m/n)$.

3 Hiding Latency in a Linear Array

In this section we present algorithm OVERLAP which simulates guest array G by host array H. Section 3.1 describes a process of killing "useless" processors of H. A processor of H is killed if it is surrounded by too much delay or if most of its neighbors are killed. Only live processors are used to carry out the simulation. Section 3.1 also considers the "computing power" of subarrays of H. In particular, labels are given to subarrays in order to indicate the number of columns of pebbles a subarray can simulate. Section 3.2 deals with the details of OVERLAP. Algorithm OVERLAP first assigns databases to live processors of H according to the labels and then carries out a simulation using redundant computation. The slowdown achieved is $O(d_{\text{ave}} \log^3 n)$, where d_{ave} is the average delay of H and n is the size of H. The algorithm is made efficient in section 3.3, and the slowdown is improved to $O(\sqrt{d_{\text{ave}}} \log^3 n)$ in section 3.4.

3.1 Killing Processors and Labeling the Tree

We create a binary tree, T, to represent the host array H. The root of T 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 in the tree corresponds to a subarray of H which contains $\frac{n}{2^k}$ processors. We refer to this subarray as a *depth k interval.* (See Figure 2.) Tree T has height $\log n$. The leaves of T correspond to single processors of H.

We now describe a procedure which *kills* "useless" processors of H. Processor p is killed if it is surrounded by too high delays or too few live processors. This is because the benefit to be gained by using p's computing power is nullified by the amount of time that it takes to communicate with p. For every depth k, we define $D_k = (\frac{n}{2^k} d_{\text{ave}})(c \log n)$ to be the "killing delay" and $m_k = \frac{n}{c 2^k \log n}$ to be the "overlap size". The constant c is specified later.

Stage 1: Killing processors that are surrounded by high delays. It can easily be seen that each processor p in H is contained in exactly one depth k interval. Call this interval I_k^p. We kill processor p if for any k, the total delay in interval I_k^p is more than D_k.

Lemma 1 *At most n/c processors are killed.*

Proof: The total delay in the array H is nd_{ave}. Hence the number of depth k intervals with delay more than D_k is at most nd_{ave}/D_k. Each depth k interval contains $\frac{n}{2^k}$ processors and so the total number of processors killed at depth k is at most $\frac{n}{c \log n}$. Hence the total number of processors killed is at most n/c. □

Stage 2: Labeling the tree and killing processors that are surrounded by few live processors. We carry out a labeling of tree T to determine if an interval has too few live processors. We first remove from T any node whose corresponding interval has no live processors. A label is attached to each of the remaining nodes as described below.

From now on we assume that v is a depth k node which has label x and corresponds to interval I. If v has two children, then we assume that they are v_1 and v_2, which have labels x_1 and x_2 and correspond to intervals I_1 and I_2 respectively. If v has one child, then we assume that it is v_1, which has label x_1 and corresponds to interval I_1. (See Figure 2.)

- Each leaf of T, which corresponds to a live processor, is given the label 1.
- The remaining nodes of T are labeled inductively. Consider a depth k node, v, which has not been removed. If v has two children in T, we give v the label $x_1 + x_2 - m_k$. If v has one child we give v the label x_1.

Lemma 2 *The label on the root of T is at least $(1 - 2/c)n$.*

Proof: From Lemma 1 at least $(1 - 1/c)n$ of the leaves are given label 1. Hence, the label of the root is at least $(1 - 1/c)n - \sum_{\text{depth } k} 2^k m_k = (1 - 2/c)n$. □

An interval is not useful if most of its processors have been killed and so we now carry out a second round of killing. If a depth k node has a label smaller than $2m_k$, we kill all the processors in its corresponding interval. We also remove from T any node whose corresponding interval has no live processors. The remaining nodes in tree T have the following properties.

Lemma 3 *Suppose v is a remaining node in T, which has depth k $(0 \leq k < \log n)$ and label x, then*

1. *$x \geq 2m_k$;*
2. *v has at least one child;*
3. *$x = x_1 + x_2 - m_k$ if v has two remaining children;*
4. *$x \leq x_1$ if v has one remaining child.*

Proof: Property 1 is immediate. Suppose v has no child removed due to stage 2 killing, then the labeling rule implies that $x = x_1 + x_2 - m_k$ if v has two remaining children and that $x = x_1$ if v has one remaining child. Otherwise assume v has child v_2 removed. Since v is remaining, the killing rule implies that $x \geq 2m_k$ and $x_2 < 2m_{k+1}$. Therefore, v must have another child v_1 labeled x_1, such that $x_1 = x - x_2 + m_k > x$. Since $x_1 > 2m_{k+1}$ and v is remaining, node v_1 is not removed. Hence, properties 2, 3 and 4 follow. □

Stage 3: Relabeling the tree. We now relabel tree T as follows. The labels are to indicate the computing power of the corresponding intervals.

- Each leaf of T, which corresponds to a processor that is not killed in stage 1 and 2, is given the label 1.
- The remaining nodes of T are labeled inductively in a similar manner to stage 2. Consider a depth k node, v, which has not been removed. If v has two children, we give v the label $x_1 + x_2 - m_{k+1}$. (Notice that the label would have been $x_1 + x_2 - m_k$ in stage 2.) If v has one child, we give v the label x_1.

By Lemma 3 and the relabeling rule, the stage 3 labels on the remaining nodes are at least as big as the stage 2 labels. Hence,

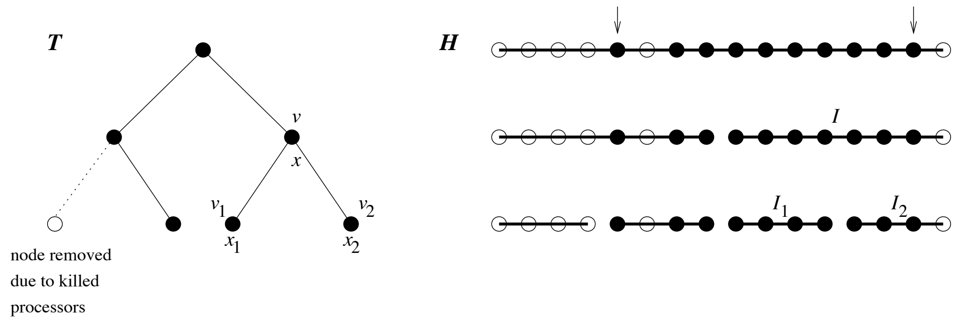


Figure 2: Nodes v, v_1 and v_2 have labels x, x_1 and x_2 and correspond to intervals I, I_1 and I_2 respectively. Live processors of H and remaining nodes of T are represented by black circles; killed processors of H and the nodes removed from T are represented by white circles. Arrows indicate the endpoints of the root interval.

Lemma 4 *After stage 3 the label on the root is at least $(1-2/c)n$, and the label on a remaining depth k node is at least $2m_k$.*

We use label x to indicate the computing power of I. That is, the corresponding interval I can simulate x columns of pebbles.

3.2 The Simulation

For clarity of presentation, we first assume that G has n' processors, where n' is the label on the root of T and the number of live processors in H. Lemma 4 implies that n' is a constant fraction of n. 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.

Assigning databases

Algorithm OVERLAP first assigns databases to the live processors of H such that the load is one. That is, each live processor of H is assigned one database. Databases $b_1, \ldots, b_{n'}$ are assigned to the live processors of H, i.e. the depth 0 interval. We assume inductively that OVERLAP assigns databases $b_{i+1}, \ldots, b_{i+x}$ to I. (Recall that node v corresponds to interval I by assumption.) If v_1 is the only child of v, then OVERLAP assigns databases $b_{i+1}, \ldots, b_{i+x}$ to I_1. If v_1 and v_2 are two children of v, then $x = x_1 + x_2 - m_{k+1}$ by the relabeling rule. OVERLAP assigns databases $b_{i+1}, \ldots, b_{i+x_1}$ to interval I_1 and $b_{i+x-x_2+1}, \ldots, b_{i+x}$ to I_2. Notice that there are m_{k+1} databases, namely $b_{i+x-x_2+1}, \ldots, b_{i+x_1}$, which are assigned to both I_1 and I_2. It is easy to see that at the leaf level each live processor of H is assigned one database, although some databases may be copied more than once.

Simulating G by H

Algorithm OVERLAP makes use of redundant computaion. In particular, if processor p of H has a copy of database b_i then p computes *every* pebble (i,t), for $t \geq 1$. OVERLAP carries out the simulation in a recursive manner. Suppose a depth k interval I is assigned databases b_{i+1}, $\ldots$, b_{i+x}. Let B_k be the box of pebbles (j,t), where $i+1 \leq j \leq i+x$ and $1 \leq t \leq m_k$. We shall show in Theorem 1 how I simulates every pebble in B_k. This simulation is done recursively by the depth $k+1$ subinterval(s) of I. At the top level of the recursion the depth 0 interval, i.e. H, simulates every pebble in B_0. (Note that B_0 contains all the pebbles created by G in the first $m_0 = \frac{n}{c \log n}$ steps.) OVERLAP then repeats to simulate the next m_0 steps of G.

Let $k_{\max} = \log n - \log\log n - \log c$. We first recursively define a set of values $s_t^{(k)}$ for $0 \leq k \leq k_{\max}$ and $1 \leq t \leq m_k$. These values correspond to the time by which a particular row of pebbles is computed.

1. Let $s_1^{(k)} = 1$ for $k = k_{\max}$.
2. Let $s_t^{(k)} = s_t^{(k+1)} + D_k$ for $1 \leq t \leq m_{k+1}$.
3. Let $s_t^{(k)} = s_{t-m_{k+1}}^{(k)} + s_{m_{k+1}}^{(k)}$ for $m_{k+1}+1 \leq t \leq m_k$.

(Recall that the total delay in a depth k interval is at most D_k by the killing in stage 1.)

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

Theorem 1 *For $1 \leq t \leq m_k$, if the value of the 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 which 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 live 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 $k+1$. Notice that the hypothesis can be applied to any depth $k+1$ interval. Let v be the corresponding depth k node of I, then v has label x. There are two cases to consider.

Case 1: Suppose that v has two children v_1 and v_2. By construction $x = x_1 + x_2 - m_{k+1}$. Let $B_{k+1} = \{(i,t) : 1 \leq i \leq x_1, 1 \leq t \leq m_{k+1}\}$. Let $y = x_1 - m_{k+1}$ and $B'_{k+1} = \{(i,t) : y+1 \leq i \leq x, 1 \leq t \leq m_{k+1}\}$. Let column C consist of pebbles (y,t) and column D consist

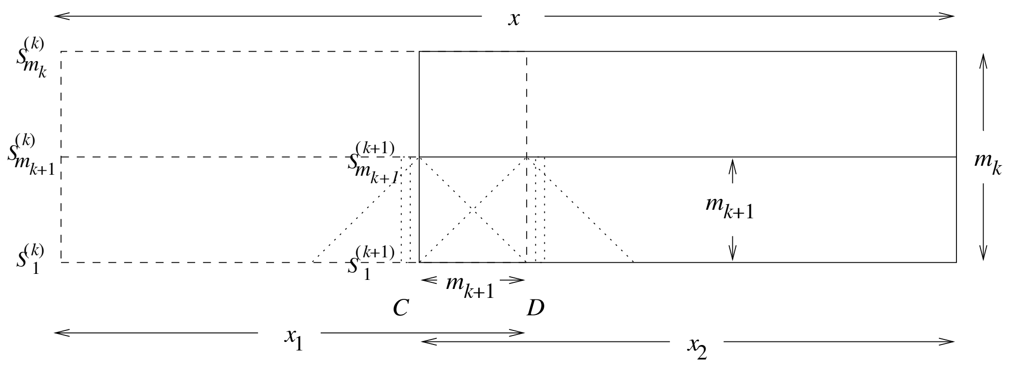


Figure 3: The box of pebbles B_{k+1}, which has size $x_1 \times m_{k+1}$, is represented by the lower left box with a dashed boundary, and B'_{k+1} is represented by the lower right box with a solid boundary. B_k is the union of all four boxes. For interval I to compute every pebble in B_k, intervals I_1 and I_2 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.

of pebbles (x_1+1,t), where $1 \le t \le m_{k+1}$. Notice 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 3.) Two facts can easily be deduced from the inductive hypothesis.

- **Fact 1** By time step $s_t^{(k+1)}$, every pebble (y,t) in column C can be computed by I_1. Notice that here there are no conditions on pebbles $(0,t)$ and (x_1+1,t), for $1 \le t \le m_{k+1}$. This is true because C and D are m_{k+1} columns apart and $x_1 \ge 2m_{k+1}$ by Lemma 4. Hence, the pebbles in column C do not depend on the pebbles $(0,t)$ and (x_1+1,t). (The dotted diagonal lines in Figure 3 show the dependencies of columns C and D.)
- **Fact 2** Let $z \ge 0$ be some constant. If the value of pebbles $(0,t)$ and (x_1+1,t) are known at time step $s_t^{(k+1)}+z$ by the left and right endpoints of interval I_1 respectively, then by time step $s_t^{(k+1)}+z$, every pebble (i,t) in B_{k+1} is computed. This is true because there is no difference between starting the simulation at time step z and at time step 0.

Similar statements can be made about the box B'_{k+1} and column D. Now suppose that the value of pebbles $(0,t)$ and $(x+1,t)$ are known at time step $s_t^{(k)}$ by the left and right endpoints of interval I respectively. Fact 1 implies 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 $s_t^{(k)}$ (cf. definition 2). Similarly, all of the pebbles in column D can be sent to the right endpoint of interval I_1 by time $s_t^{(k)}$. Notice that if $1 \le t \le m_{k+1}$ then $s_t^{(k)}$ is greater than $s_t^{(k+1)}$ by a constant amount, namely D_k. Hence, Fact 2 implies that if pebble (i,t) is in either box B_{k+1} or B'_{k+1} then (i,t) is computed by time $s_t^{(k)}$. (If (i,t) is in the overlap then it is computed by both intervals by time $s_t^{(k)}$.) This implies that if pebble (i,t) is in the bottom half of box B_k then it is computed by time $s_t^{(k)}$.

Once the bottom half has been simulated interval I simulates the top half of box B_k in a similar manner. Thus, if pebble (i,t) is in the top half, then it is computed by time $s^{(k)}_{m_{k+1}} + s^{(k)}_{t-m_{k+1}}$ which equals $s_t^{(k)}$ (cf. definition 3). 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 $s_t^{(k)}$.

Case 2: The case in which v has one child is simpler. Let v_1 be the child of v. By construction, v_1 has label $x_1 = x$. By the induction hypothesis and Fact 2, if the values of the pebbles $(0,t)$ and (x_1,t), for $1 \le t \le m_{k+1}$, 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 live processors (and hence the same endpoints), the above statement holds for I. Interval I then computes the top half in the same manner. Thus if pebble (i,t) is in the top half then it is computed by time $s^{(k)}_{m_{k+1}} + s^{(k)}_{t-m_{k+1}}$ which equals $s_t^{(k)}$ (cf. definition 3). The inductive step is thus complete. □

Recall that n' is the label of the tree root and n' is a constant fraction of n by Lemma 4. We have the following.

Theorem 2 *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 computed by G in the first $m_0 = \frac{n}{c \log n}$ time steps. The interval I_0 corresponding to the root contains all the live processors of H. Since pebbles $(0,t)$ and $(n'+1,t)$ are available at time step 0 by assumption, Theorem 1 implies that I_0, which is H, computes the pebbles in box B_0 by time $s^{(0)}_{m_0}$. From the definitions we have the recurrence $s^{(k)}_{m_k} = 2s^{(k+1)}_{m_{k+1}} + 2D_k$. Since $D_k = (\frac{n}{2^k} d_{\text{ave}})(c \log n)$, it follows that $s^{(0)}_{m_0} = 2^k s^{(k)}_{m_k} + 2kD_0$, for $k > 0$. Hence, by setting k to $k_{\max}$ we obtain $s^{(0)}_{m_0} \le \frac{n}{c \log n} + 2cd_{\text{ave}} n \log^2 n = O(d_{\text{ave}} n \log^2 n)$, giving a slowdown of $O(d_{\text{ave}} \log^3 n)$. □

Remarks. It is clear that the bandwidth required for the communication between depth k intervals is at most the bandwidth of G. Hence, our assumption that the bandwidth on H is at most $\log n$ times the bandwidth on G is enough to ensure that we can carry out all the communication. This assumption can be removed if we are willing to pay an extra factor of $\log n$ in the slowdown. Also notice that our simulation works for any constant $c > 2$.

3.3 A Work Efficient Algorithm

In this section we modify OVERLAP so that the algorithm is work efficient. Suppose that host network H is a linear array of n processors with average delay d_{ave}, and guest network G is a linear array of $d_{\text{ave}} n \log^3 n$ processors. The killing of processors in H and the labeling of tree T is carried out exactly as before. If database b_i is assigned to processor p in the previous discussion, OVERLAP now assigns $b_{(i-1)\alpha\beta+1}, \ldots, b_{i\alpha\beta}$ to p, where α is some constant and $\beta = d_{\text{ave}} \log^3 n$. Hence, the load on H is $O(d_{\text{ave}} \log^3 n)$. Algorithm OVERLAP performs the simulation recursively as in Theorem 1. The only difference is that at depth $k = k_{\max}$ each live processor computes $\alpha d_{\text{ave}} \log^3 n$ pebbles. Therefore, $s_1^{(k)} = \alpha d_{\text{ave}} \log^3 n$. It is easy to check that the recurrence $s_{m_k}^{(k)} = 2 s_{m_{k+1}}^{(k+1)} + 2D_k$ and its solution $s_{m_0}^{(0)} = 2^k s_{m_k}^{(k)} + 2kD_0$ remain true. Hence, by setting k to $k_{\max}$ we obtain $s_{m_0}^{(0)} = O(d_{\text{ave}} n \log^2 n)$, giving a slowdown of $O(d_{\text{ave}} \log^3 n)$. We have,

Theorem 3 *Suppose that H is an n-processor linear array with average delay d_{ave} and G is a $(d_{\text{ave}} n \log^3 n)$-processor linear array with unit delay links. Algorithm* OVERLAP *achieves a slowdown of $O(d_{\text{ave}} \log^3 n)$ and a load of $O(d_{\text{ave}} \log^3 n)$ when simulating G by H. This simulation is work efficient.*

3.4 Improving the Slowdown

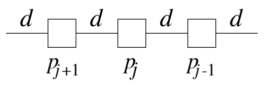
We show below that the slowdown can be improved by a factor of $\sqrt{d_{\text{ave}}}$. We first consider the special case when all the delays on the host links are the same.

Theorem 4 *Let H_0 be an n-processor linear array in which each link has delay d. Let G be an $n\sqrt{d}$ processor linear array in which each link has delay 1. Then G can be simulated on H_0 with slowdown $O(\sqrt{d})$. The simulation is work efficient and has minimum load.*

Proof: We show how to simulate $\sqrt{d}$ steps of G in $5d$ steps on H_0. For $1 \le j \le n$ let,

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

$$\begin{aligned}
A &= \{\text{Pebbles } ((j-2)\sqrt{d}, t) : && 1 \le t \le \sqrt{d}\}, \\
B &= \{\text{Pebbles } ((j-1)\sqrt{d}+1, t) : && 1 \le t \le \sqrt{d}\}, \\
C &= \{\text{Pebbles } ((j\sqrt{d}, t) : && 1 \le t \le \sqrt{d}\}, \\
D &= \{\text{Pebbles } ((j+1)\sqrt{d}+1, t) : && 1 \le t \le \sqrt{d}\}.
\end{aligned}$$



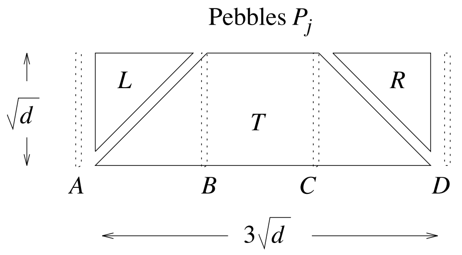


Figure 4: Simulating $\sqrt{d}$ steps of G on H_0.

(See Figure 4.) Processor p_j of H_0 computes all the pebbles in P_j. Notice that this set of pebbles has overlaps with the pebbles in P_{j-1} and P_{j+1}. First processor p_j computes the pebbles in the trapezium, T. There are $2d$ pebbles in T and so this takes $2d$ steps. Processor p_j then passes column B to processor p_{j-1} and receives column A from p_{j-1}. It also passes column C to processor p_{j+1} and receives column D from p_{j+1}. This communication takes $d + \sqrt{d} < 2d$ steps using pipelining. Processor p_j can now compute the pebbles in triangles L and R in d steps. Hence, it takes at most $5d$ steps in total for processor p_j to compute every pebble in P_j. Once this is done the next $\sqrt{d}$ time steps can be simulated in a similar fashion. □

In [2], $\Omega(\sqrt{d})$ is proved to lower bound the slowdown of simulating G on H_0.

Combining Theorem 2 and 4 we can improve the slowdown to $O(\sqrt{d_{\text{ave}}} \log^3 n)$ while preserving efficiency and minimum load. Suppose that the guest G is a linear array of $\sqrt{d_{\text{ave}}} n \log^3 n$ processors and the host H is a linear array of n processors with average delay d_{ave}. We make use of an intermediate network H_0, which is a linear array of $n \log^3 n$ processors and has a delay of d_{ave} on every link. Theorem 4 implies that network H_0 can simulate G with a slowdown of $O(\sqrt{d_{\text{ave}}})$. Theorem 2 implies that H can simulate H_0 with a slowdown of $O(\log^3 n)$. The combined slowdown is thus $O(\sqrt{d_{\text{ave}}} \log^3 n)$. The minimality of the load is immediate from the database assignment. Therefore,

Theorem 5 *An n-processor host linear array of average delay d_{ave} can simulate a $(\sqrt{d_{\text{ave}}} n \log^3 n)$-processor guest linear array with a slowdown of $O(\sqrt{d_{\text{ave}}} \log^3 n)$ and a load of $O(\sqrt{d_{\text{ave}}} \log^3 n)$. The simulation is work efficient and has minimum load.*

4 Simulating Linear Arrays on General Networks

Algorithm OVERLAP can be generalized to the simulation of a guest linear array by an arbitrary bounded-degree fixed-connection host network. The following fact [8] is used in our argument.

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

Let $\mathcal{H}$ be the n-node linear array embedded in H by Fact 3. The proof of Fact 3 [8, page 470] implies that if H has bounded degree δ then $\mathcal{H}$ has average delay at most δd_{ave}. By Theorem 5, $\mathcal{H}$ can simulate G with a slowdown of $O(\sqrt{d_{\text{ave}}} \log^3 n)$. Thus,

Theorem 6 *A connected bounded-degree n-node network H with average delay d_{ave} can simulate a $(\sqrt{d_{\text{ave}}} n \log^3 n)$-node linear array G with a slowdown of $O(\sqrt{d_{\text{ave}}} \log^3 n)$ and a load of $O(\sqrt{d_{\text{ave}}} \log^3 n)$.*

Theorem 6 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 2 adjacent cliques has delay n, then H has $d_{\text{ave}} < 4$. Suppose m connected cliques are used to simulate n steps of G. A work argument implies a slowdown of at least $\sqrt{n}/m$. A linear array embedded in these m connected cliques has a total delay of at least mn, which implies a slowdown of m. Hence, the slowdown is at least $\max\{\sqrt{n}/m, m\} \geq n^{1/4} = \omega(\sqrt{d_{\text{ave}}} \log^3 n)$, since the average delay is a constant.

5 Simulating Other Networks on NOWs

The methods described in sections 3 and 4 can be used to simulate a variety of other guest network computations on an arbitrary NOW in a latency-hiding fashion. As an example of how to apply the methods, we show in what follows how to simulate an $m \times m$ array G on an n-node host H. As before, we assume G has unit delay on all the links and H has bounded degree and average delay d_{ave}. As discussed in section 4 there exists a linear array $\mathcal{H}$ such that $\mathcal{H}$ is embeded 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 H_0, where H_0 has $n_0 = n \log^3 n$ processors and delay d_{ave} on all the links.

Theorem 7 *Let $m = d_{\text{ave}}^{1/3} n_0^{2/3}$. We can simulate an $m \times m$ array G on H_0 with slowdown $O(m + m^2/n_0)$.*

Proof: The simulation depends on the relative sizes of d_{ave} and n_0.

Case 1: If $d_{\text{ave}} < n_0$, then $m < n_0$. Each of the first m processors of H_0 simulates one column of G; the other processors of H_0 are not used. To simulate one step of G, a processor of H_0 computes m pebbles and then communicates with each of its neighbors. This takes $m + d_{\text{ave}}$ steps, which is smaller than $2m$ steps. Hence the slowdown is $O(m)$.

Case 2: If $d_{\text{ave}} \geq n_0$, then $m \geq n_0$. Each processor of H_0 simulates m/n_0 columns of G. To simulate m/n_0 steps of G, each processor of H_0 computes at most $(3m/n_0)(m/n_0)m$ pebbles and then communicates with each of its neighbors. This takes $3m^3/n_0^2 + d_{\text{ave}} = 4m^3/n_0^2$ steps. Hence the slowdown when simulating every m/n_0 steps is $O(m^2/n_0)$. $\square$

Theorem 6 implies that $\mathcal{H}$ (and thus H) can simulate H_0 with a slowdown of $O(\log^3 n)$. Combined with Theorem 7, the total slowdown is $O\left((m + m^2/n_0)\log^3 n\right)$, which is $O(m \log^3 n + m^2/n)$.

Theorem 8 *Suppose that H is an n-processor bounded-degree host network which has average delay d_{ave}, and G is an $m \times m$ guest array, where $m = d_{\text{ave}}^{1/3} n^{2/3} \log^2 n$. Network H can simulate G with a slowdown of $O\left(d_{\text{ave}}^{1/3} n^{2/3} \log^5 n + d_{\text{ave}}^{2/3} n^{1/3} \log^4 n\right)$.*

Stated differently, the slowdown used to simulate an N-node 2-dimensional array on a NOW with average delay d_{ave} is $O(\sqrt{N} \log^3 N + N^{1/4} \sqrt{d_{\text{ave}}} \log^3 N)$. Theorem 8 can be generalized to higher dimensional arrays. Whether or not these bounds are tight up to a polylog factor remains an interesting open question.

6 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. We suspect that our technique can be generalized to the case in which each database has a constant number of 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. The delays on H_1 are as follows. Every $\sqrt{n}$-th link of H_1 has a delay of $\sqrt{n}$ and all other links have unit delay. 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 $d_{\text{max}} = \sqrt{n}$, whereas d_{ave} is a constant. (Note that the above argument makes no assumption on the load.) Thus,

Theorem 9 *If each database can have at most one copy, the slowdown when simulating G by H_1 is d_{max}.*

For the case in which each database is allowed at most two copies we construct a host network H_2 whose average delay is $O(1)$, but for which the slowdown when H_2 simulates G is $\Omega(\log n)$. Network H_2 is made up of $\Theta(n)$ processors and the link delays are either 1 or d, where $d = \sqrt{n}$. The following is a recursive construction of H_2 in which we define a series of boxes. (See Figure 5.) We regard H_2 as a level k box, where $k = \log(n/d)$. Network H_2 consists of two level $k-1$ boxes, which are connected by $2^k d/\log n$ edges of delay 1. In general, a level ℓ box, for $1 \leq \ell \leq k$, consists of two level $\ell - 1$ boxes, which are connected by $2^\ell d/\log n$ edges of delay 1. We say that these $2^\ell d/\log n$ processors are in a *segment.* A level 0 box consists of a single edge of delay d. Notice that a level ℓ box contains 2^ℓ edges of delay d and $2^\ell d\ell/\log n$ edges of delay 1. Hence, H_2 has $\Theta(n)$ processors and constant average delay d_{ave}. The following property of H_2 is used in the proof of the lower bound.

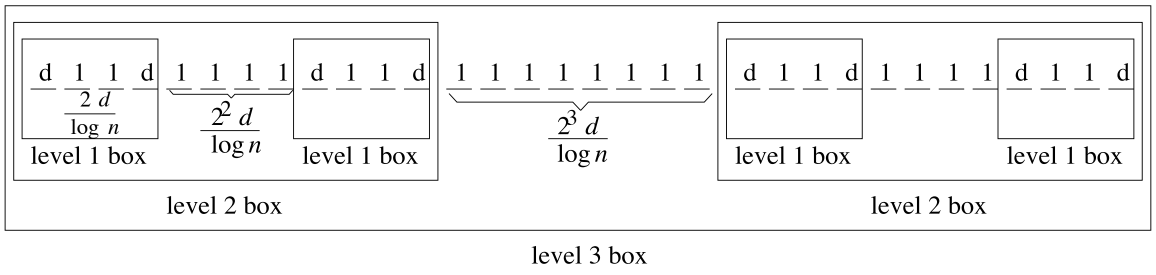


Figure 5: A level 3 box. Host network H_2 is a level k box, where $k = \log(n/d)$.

- **Fact 4** If processors p and q are in two different segments I and J, then the delay between p and q is at least $\min\{u \log n, v \log n\}$, where u and v are the numbers of processors in segments I and J respectively. (Notice that u and v have to have the form $\frac{2^\ell d}{\log n}$.) In particular, the delay between p and q is at least $d > \log n$.

Theorem 10 *If each database is allowed at most two copies and the load is a constant c, then the slowdown when simulating G by H_2 is $\Omega(\log n)$.*

Proof: There are two cases to consider.

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

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

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

Notice that the path goes backwards in time, and that the path zigzags during time steps k, for $k \in B \cup C \cup E \cup F$. (See Figure 6.)

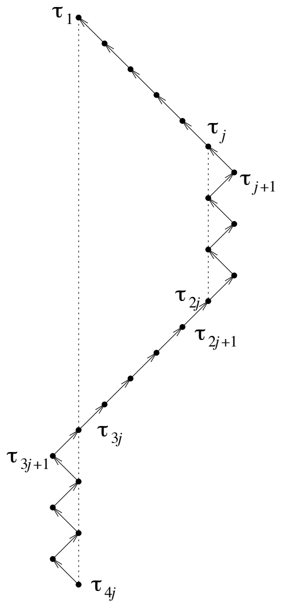


Figure 6: A path of $4j$ pebbles, where j is even.

By assumption processors p_k, for $k \in C \cup E$, can only belong to segment I or J. If processors p_k, for $k \in C \cup E$, do not belong to the same segment, then Fact 4 implies a delay of $(j/c) \log n$ for the communication between segments I and J. Hence, it takes more than $(j/c) \log n$ steps to compute this path of $4j$ pebbles.

If processors p_k, for $k \in C \cup E$, all belong to segment I, then Fact 4 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. Simiarly, 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}, and S_J be the sequence of consecutive databases which includes b_{i-1} and which all have copies 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 which 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 Fact 4. Therefore, the slowdown is $\Omega(\log n)$. □

7 Open Questions

This paper leaves many interesting questions unanswered. Most generally, how much slowdown is needed to efficiently simulate a guest network G with unit delay links on a host NOW H with arbitrary link delays? In the case that G is a ring, we have proved upper and lower bounds that are existentially tight to within a polylogarithmic factor. (For lower bounds, see [2].) It would be nice to close the gap between the upper and lower bounds. More importantly, it would be nice to devise an approximation algorithm that was *always* within a polylogarithmic factor of optimal for any NOW. In the case that G is a 2-dimensional array, our bounds may be even less tight (although it may be possible to show that the bounds are existentially tight up to polylogarithmic factors — we are still investigating this issue). The case when G and H are both 2-dimensional arrays is also very intriguing but currently beyond our abilities. More generally, it would be interesting to consider the case when G and H have identical network structures (but different link delays) in order to study the effect of latencies in isolation.

Ultimately, one is interested in simulating efficiently types of networks that appear often in the architectures of parallel computers, like trees, arrays, butterflies and hypercubes, on a network of workstations with arbitrary link delays.

Acknowledgements

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

References

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

[2] M. Andrews, T. Leighton, P. T. Metaxas, and L. Zhang. Automatic methods for hiding latency in high bandwidth networks. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing* (to appear), 1996.

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

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

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

[6] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. 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.

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

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

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

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

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

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

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

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

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

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