Automatic Methods for Hiding Latency in Networks of Workstations *

Matthew Andrews† Lisa Zhang‡

The Problem Most algorithms for parallel and distributed computation are designed for a model in which all the links have unit delay. In reality, there are often substantial delays associated with some or all of the links. They delays cause latencies which can slow down the entire computation. In this abstract we present automatic methods for mitigating the impact of latency in networks of workstations (NOWs). Formally, we consider two networks, the *guest* network G which has unit-delay links and the *host* network H which has arbitrary delays on its links. In a *simulation* of G on H, network H carries out the same step-by-step computations as G when G is used in a general purpose way. Our goal is to minimize the *slowdown* of the simulation. Our approach is similar in spirit to the "complementary slackness" method of latency hiding, (i.e. loading processors with enough work while waiting for communication), 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.

The Models We device methods for automatic latency hiding under two models, the *dataflow* model and the *database* model. We use *pebbles* to represent the computations performed by the guest processors. Consider the example of a guest ring. In the dataflow model pebble (i, t) records the computation performed by processor g_i of G at time step t, and pebble (i, t) depends on pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$. In a simulation of G on H, host processors must compute *every* pebble created by G. This model is sufficiently general so as to be applicable for many algorithms such as matrix operations, Fourier transformations, etc. However, it does not include applications where the computation depends on the state of a large local memory or database.

By contrast, in the database model each processor g_i of G owns a database b_i. Pebble (i, t) records not only the computation by g_i at time t but also the changes to database b_i incurred by this computation. Processor g_i consults b_i before each computation and updates b_i after each computation. Hence, pebble (i, t) depends on pebbles $(i-1, t-1)$, $(i, t-1)$ and $(i+1, t-1)$ as well as the current state of b_i. Host processors copy the databases *before* the simulation of G. Because of its large size, a database cannot be passed through the limited-bandwidth links during the simulation. If processor p of H is to compute pebble (i, t), then p must have an updated copy of database b_i.

Results – Dataflow model

G	H	Slowdown
Ring (Linear array)	Connected and bounded-degree	$O(\sqrt{d_{\text{ave}}})$
2-D array	2-D array	$O(d_{\text{ave}}^{2/3} \log^{5/3} N)$

Database model

G	H	Slowdown
Ring (Linear array)	Connected and bounded-degree	$O(\sqrt{d_{\text{ave}}} \log^3 N)$
2-D array	2-D array	open
2-D array	Connected and bounded-degree	$O(d_{\text{ave}}^{1/3} N^{2/3} \log^5 N + d_{\text{ave}}^{2/3} N^{1/3} \log^4 N)$

Host H is an N-processor network which has an average delay d_{ave}. Guest G has unit-delay links and its size can be chosen such that the simulation is work-efficient. Prior approaches would require slowdown d_{max}, where d_{max} is the maximum delay of H. Our improvement is particularly impressive when $d_{\text{max}} \gg d_{\text{ave}}$, which is often the case for NOWs.

Techniques for the dataflow model

Rings. Simulating a ring and simulating a linear are essentially the same, since a linear array can simulate a ring with a slowdown of 2. We first consider the case in which G and H are n-processor linear arrays. If each host processor is to simulate a column of pebbles (i, t), for $t \geq 0$, then a communication bottleneck occurs. Instead, processor p_i of H simulates a diagonal stripe of pebbles $(i - t + 1, t)$ for $1 \leq t \leq i$ and $t \leq n/2$. (See Figure 1.) In the original computation by G, processor g_i depends on both g_{i-1} and

*This research is joint work with Professor Tom Leighton and Professor Takis Metaxas. For more details see http://theory.lcs.mit.edu/~andrews.

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

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

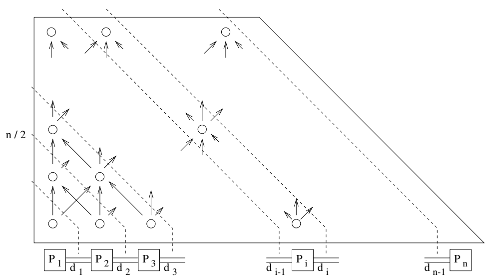

Figure 1: Each host processor simulates a diagonal stripe of pebbles.

g_{i+1}. In the simulation by H, p_i depends on p_{i-1} and p_{i-2}. Hence, a process which involves two-way communication is transformed into a process which involves only one-way communication. Processor P_i is able to compute its stripe after a delay of $\sum_{k=1}^{i-1} d_k$. Hence H computes all the pebbles in the trapezoid in Figure 1 in $n/2 + \sum_{k=1}^{n-1} d_k$ steps. By computing a symmetric trapezoid in a similar fashion, H finishes simulating the first $n/2$ steps of G in $n+2\sum_{k=1}^{n-1} d_k$ steps, implying a slowdown of $O(d_{\text{ave}})$. The next $n/2$ steps are then simulated.

When H simulates a larger linear array of m processors (i.e. each host processor simulates m/n stripes), the resulting slowdown is $O(m/n + d_{\text{ave}}n/m)$. For $m = n\sqrt{d_{\text{ave}}}$, the slowdown is improved to $O(\sqrt{d_{\text{ave}}})$.

For a connected, bounded-degree network H of n processors, there exists an n-processor linear array, L, which is a subgraph of H and has average delay $O(d_{\text{ave}})$. Host H uses L to simulate G.

2-D arrays. Simulating a 2-D array on a 2-D array is more complicated. Let H and G be arrays of dimension $n \times n$. If the above technique is applied directly, the resulting slowdown is $\Theta(D/n)$, where D is the length of the longest monotone path from processor $p_{1,1}$ to $p_{n,n}$ of H. (Monotone paths travel in two directions, up and right. The length of a path is the total delay on it.) However, $\Theta(D/n)$ can be $\omega(d_{\text{ave}})$.

To overcome this problem, we aim to find an embedding from G into H such that the following holds. Firstly, O(1) guest processors are embedded to each host processor. Secondly, the image of any monotone path of G in H is "short". (Note that the image of a monotone path does not need to be monotone.) The embedding is achieved in two stages. In stage 1, host array H is divided into four arrays recursively. At each level of the recursion, "useless" processors of H are killed. A host processor is *useless* if it is surrounded by high delays or by many killed processors. A constant fraction of the host processors are killed. Only live processors are used to carry out the simulation. In stage 2, guest processors are mapped to live host processors such that only a "small" delay is picked up at each level of recursion. Our embedding algorithm yields an $O(d_{\text{ave}} n \log^{5/2} n)$ delay on the longest monotone path. This gives a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$. When host H of size $n \times n$ simulates G of a larger size, $(d_{\text{ave}}^{1/3} n \log^{5/6} n) \times (d_{\text{ave}}^{1/3} n \log^{5/6} n)$, the slowdown can be improved to $O(d_{\text{ave}}^{2/3} \log^{5/3} n)$. It remains an open question if this bound is tight.

In the case when the delays of H are randomly arranged, the longest monotone path has length $O(d_{\text{ave}}n)$ with high probability. Hence, the slowdown is $O(d_{\text{ave}})$ with high probability. Again, $O(d_{\text{ave}})$ can be improved to $O(d_{\text{ave}}^{2/3})$.

Techniques for the database model. It is easier to overcome latencies in the dataflow types of computations than in computations that require access to large local databases. Intuitively, 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 many copies of a database updated. If each database is only allowed one copy, then the slowdown can be as much as $\Theta(d_{\max})$, whereas $d_{\text{ave}} = O(1)$. If each database is allowed two copies and each host processors is allowed to copy $O(1)$ databases, then the slowdown can be $\Theta(\log n)$, whereas $d_{\text{ave}} = O(1)$.

For an n-processor host linear array and an m-processor guest linear array, we present a simulation that achieves *minimum load*, i.e. each host processor has a copy of $O(m/n)$ databases. The simulation begins with killing useless processors of H in a similar manner to the dataflow model. The live processor of H carry out the simulation recursively and make nontrivial use of redundant computation. Host processors first copy databases in an "overlapping" fashion, i.e. neighboring live processors copy some databases in common. Host processors then recursively interleave computing new pebbles with recomputing pebbles (in order to update the databases). When $m = \sqrt{d_{\text{ave}}} n \log^3 n$, the slowdown is $O(\sqrt{d_{\text{ave}}} \log^3 n)$.

It remains an open question if the bounds for the dataflow model can be improved. Our ultimate goal is to efficiently simulate types of networks that appear often in the architectures of parallel computers (e.g. trees, arrays, hypercubes) on an unstructured network of workstations with arbitrary link delays.