p045.ps: Efficient Automatic Simulation of Parallel Computation on..., DO NOT DISTRIBUTE!!!

Efficient Automatic Simulation of Parallel Computation on Networks of Workstations

Christos Kaklamanis * Danny Krizanc † Manuela Montangero‡
Pino Persiano§

January 20, 2000

Abstract

Andrews *et al.* [1, 2] introduced a number of techniques for automatically hiding latency when performing simulations of networks with unit delay links on networks with arbitrary unequal link delays. In their work, they assume that processors of the host network are identical in computational power to those of the guest network being simulated. They further assume that the links of the host are able to pipeline messages, i.e., they are able to deliver P packets in time $O(P+d)$ where d is the delay on the link.

In this paper we examine the effect of eliminating one or both of these assumptions. In particular, we provide an efficient simulation of a linear array of homogeneous processors with unit link delays on a linear array of heterogeneous processors with arbitrary link delays and we show that the slowdown achieved by our simulation is optimal. We also consider the case of simulating a clique of homogeneous processors with unit link delays on a clique of heterogeneous processors with arbitrary delay reducing the slowdown from the obvious bound of the maximum link delay to the average of the link delays. For the case of the linear array we consider both links with and without pipelining. For the clique simulation the links are not assumed to support pipelining.

The main motivation of our results (as was the case with Andrews *et al.*) is to mitigate the degradation of performance when executing parallel programs designed for different architectures on a Network of Workstations (NOW). In such a setting it is unlikely that the links provided by the NOW will support pipelining and it is quite probable the processors will be heterogeneous. Combining our result on clique simulation with well-known techniques for simulating shared memory PRAMs on distributed memory machines provides an effective automatic compilation of a PRAM algorithm on a NOW.

1 Introduction

In this paper we consider the problem of executing parallel programs designed in one setting (e.g. for a homogeneous array of processors or PRAM) in an entirely different one (e.g., a Network of Workstations (*NOW* for short)). A NOW is a very attractive and widely available distributed system found in university departments, software houses, etc; even a co-operating subset of the Internet may be thought of as a NOW. In many situations the workstations remain idle for significant periods of time. By harnessing their computational power when their owner is not using them (*e.g.*, at night or during weekends), they form a valid alternative to parallel machines for executing parallel programs.

The main problem we deal with here is to determine the degradation of the performance of the algorithm introduced by the simulation of one architecture on another. This is a classical problem in the theory of parallel algorithms and several solutions have already been proposed including the use of redundant computation [4, 6, 7] and complementary slackness [3, 5, 6, 8, 9, 10, 11, 12]. These have been shown effective for hiding queueing and congestion delays introduced by the links of distributed systems. While these approaches have been adopted in some special cases with success, they all have an undesirable characteristic: it is always the programmer

*Computer Technology Institute and Dept. of Computer Engineering and Informatics, University of Patras 26500 Rio, Greece
†Mathematics Department, Wesleyan University, Middletown CT 06459, USA
‡Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (Salerno), Italy.
§Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (Salerno), Italy.

that has to tailor the parallel algorithm to the specific distributed architecture on which the algorithm will be performed and look for an *ad hoc* simulation.

Automatic latency hiding. Andrews *et al.* in [1, 2] introduced the possibility of automatically determining the simulation once the parallel algorithm and the structure of the host network (e.g., a NOW) are known. This approach is interesting because it moves the problem of adjusting programs for the specific parallel architecture of interest from software developers to compilers or to run-time libraries. In fact, algorithm designers and software developers can, respectively, design parallel algorithms and develop software for distributed systems assuming unitary delays on links and identical processors; then, once the characteristics of the NOW on which to run the software are known, the software is automatically compiled for the current architecture. Moreover, whenever the underlying NOW changes, with minimal effort the code can be recompiled and the same algorithm can be simulated on a different NOW with no need to rewrite code.

Andrew *et al.* concentrated their attention on parallel algorithms for linear arrays, automatically simulating them with a NOW with an embedded array structure. Their setting is the following: two n processor arrays G, the *guest*, and H, the *host*, are given. G has unit delays on its links while H has arbitrary delays $d_0, \ldots, d_{n-1} > 1$ on its links and average delay d_{ave}. They consider the case in which all processors of the host array have the same computational power as the guest array processors and as each other, i.e., the processors are homogeneous. Furthermore, they assume that links of the host can pipeline messages, *i.e.*, a link with delay d can be seen as a chain of d unitary links connected by gates that can receive and send a message at each instant of time. With such a link model, at each instant of time a new message can be sent on a link and, after the first d instants of time, the message can be picked up at the other end of the link. Thus P messages can be injected in P consecutive step into a link, and the last one is received after $P + d$ steps. They distinguish between two different models: the *dataflow model* and the *database model*. In the dataflow model the computation performed by a processor p at step t depends only on the results of the computation performed by p and its neighbors at step $t-1$. In the database model each processor has its own database and at each computation step a processor reads its memory and the messages received from its neighbors and possibly updates its database. The size of the databases makes it infeasible for two processors to exchange databases once the simulation has started and only updates to the databases may be exchanged. Andrews *et al.* in [1] showed that in the dataflow model a linear array with arbitrary delay can simulate a linear array with unitary delays with a slowdown of $O(\sqrt{d_{ave}})$. In [2] they show that in the database model a slowdown of $O(\sqrt{d_{ave}} \cdot \log^3 n)$ can be achieved.

We believe that the assumptions of homogeneous processors and pipelined links used in [1, 2] are too restrictive. In general in a NOW there are no constraints on the relative speed of computation of the individual workstations. Moreover, some amount of pipelining on links may be appropriate in some situations (e.g., where delay is dominated by processing or queuing delays on multiple physical connections between workstations) but not in most situations, and in particular in the standard NOW setting of workstations belonging to local network. In this case, if P messages are sent over a link, we expect it will take time $O(P \cdot d)$ to deliver all of them.

Summary of results. In this paper we extend the work of [1, 2] by presenting simulations for parallel algorithms designed for arrays and cliques for the cases in which processors have different speeds and/or the links do not allow pipelining.

In the first part we give a simulation of a computation of a linear array of homogeneous processors connected by unit-delay links on a linear array of heterogeneous processors connected by links of arbitrary delays in the dataflow model. We also show that the slowdown achieved by our simulation is optimal. We consider both the case of links that allow pipelining and links that do not allow pipelining. These results are easily extended to the case where the host network is an arbitrary bounded degree network using the same embedding technique used in [1, 2].

In the second part, we consider simulations of cliques by cliques in the database model. We do not assume that links can pipeline messages and we analyse both the cases in which processors are homogeneous and heterogeneous. In in the first case we achieve a slowdown that is proportional to the average link delay. In the second case, the slowdown is adjusted by a factor related to the computational speed of the host processors.

The results of the second part combined with well-known techniques for simulating shared memory on distributed memory architectures (see [12] for references) yields an efficient automatic method of compiling

PRAM algorithms on a NOW. By considering PRAMs and allowing compilation to specific architectures has the advantage of completely freeing algorithm designers and software developers from considerations relative to the topological structure of the underlying network of workstation.

2 Simulating linear arrays

In this section we present our results about the simulation of linear arrays by NOWs with underlying linear array topology.

2.1 Links with pipelining

In this section we give a simulation and matching lower bound in the dataflow model for the case of a heterogeneous linear array of processors with links that allow pipelining of messages.

We are given an array G of n processors p_i, $i = 0, \dots, n-1$. Processor p_i, $0 < i < n-1$, can communicate with its two neighbours p_{i-1} and p_{i+1}, while processor p_0 can communicate only with p_1 and p_{n-1} only with p_{n-2}. Links between processors have unit delay, *i.e.*, a message sent on a link needs one unit of time to arrive to its destination. A computation of G, of length T, naturally defines a DAG with vertices (x, y), for $x = 0, \dots, n-1$ and $y = 0, \dots, T$, representing the computation performed by processor p_x at time step y. The computation (x, y) depends on the outcome of the computation of p_x and its neighbours at the previous time step, thus (x, y)'s incoming edges are: $((x+1, y-1), (x, y)), ((x, y-1), (x, y)), ((x-1, y-1), (x, y))$.

Our aim is to simulate a computation on G using a host array H of $m \leq n$ processors that communicate through links with delay $d > 1$. More precisely, d_i is the delay on the link connecting p_i to p_{i+1}. Processors of H have different computational power. We associate with each processor p_i its speed s_i, meaning that in one step processor p_i can simulate s_i steps of a processor in G.

A first attempt to simulate G with H could be to slow down every processor to the calculation speed of the slowest in H and to think that every communication requires the maximum delay on H. On the contrary, in the following, to hide the latency introduced by non unitary delays we take advantage of the fact that some processors are more powerful than others.

The techniques presented in this section resemble those of [1] used for the case of an array of homogeneous processors.

2.1.1 The stripe algorithm

Consider the first n steps of G's computation. Define L as the triangle formed by the pebbles (j, t) of the DAG such that $j \leq n - t$ and R as the one formed by pebbles (j, t) such that $j \leq t$. The algorithm first simulates the first $n/2$ steps of L, then the first $n/2$ steps of R; in this way every pebble of the first $n/2$ steps are simulated. In the same way it will simulate the following steps of the computation, $n/2$ at a time.

Only a portion of array H is used to perform the simulation; w.l.o.g. we suppose we use $m_I \leq m$ processors in the interval $I = \{p_0, \dots p_{m_I}\} \subseteq H$.

We will use the following notation:

$$S_i = \sum_{j=0}^{i} s_j, \;\; S_I = \sum_{p_j \in I} s_j \text{ and } D_i = \sum_{j=0}^{i-1} d_j.$$

Let k be the minimum integer such that $n \leq \sum_{j=0}^{m_I - 1} k s_j$. It is easy to see that $k \leq 1 + n/S_I$.

To simulate the bottom half of L, we divide it into slanting stripes and each processor p_i computes all pebbles in stripe R_i, which is defined in the following way:

$$R_i = \{(j, t) \mid t = 0, \dots, \min\{n/2 - 1, kS_{i-1}\} \text{ and } j = \max\{0, kS_{i-1} - t\}, \dots, kS_i - 1 - t\}\,.$$

Every stripe is computed row by row in a bottom-up manner and every row is computed from left to right.

Lemma 1 *Processor p_i $(0 \leq i \leq m_I - 1)$ computes pebble $(kS_i - t, t)$ at time step less or equal to $k(t+1) + D_i$.*

Proof. First observe that the computation of processor p_i depends on pebbles calculated by p_{i-1} and possibly, when $k = 1$ and $s_{i-1} = 1$, by p_{i-2}. Then, observe that, if p_i did not have to wait for information from its neighbours, it could compute all the pebbles on a row of its stripe, from $(kS_{i-1} - t, t)$ to $(kS_i - t - 1, t)$, in k units of time.

The proof is by induction on i. The base case for p_0 follows easily by observing that its computation never needs information from other processors and that p_0 needs at most k units of time to compute every row. Thus, pebble $(kS_0, 0)$, the last of row 0, is done at time k;$(kS_0 - 1, 1)$, the last of row 1, is done at time $2k$ and so on.

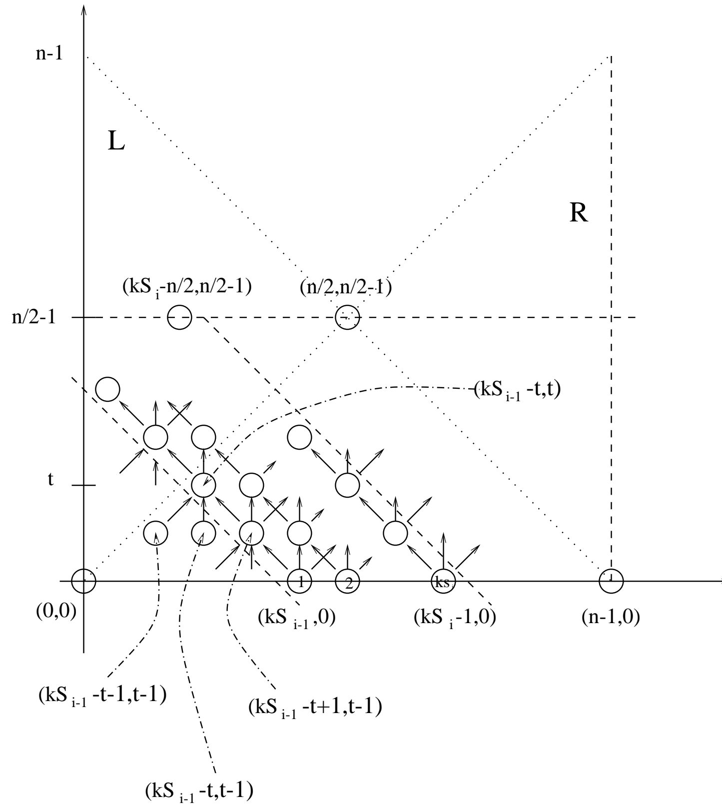

Figure 1: Processor p_i computes a stripe of pebbles (j, t) such that $kS_{j-1} + (1 - t) \leq j \leq kS_j - t$, every row is computed left to right and rows are computed in a bottom-up manner. To compute pebble $(kS_{i-1} + (1 - t), t)$ processor p_i needs pebbles $(kS_{i-1} + (2 - t), t - 1)$, $(kS_{i-1} + (1 - t), t - 1)$ and $(kS_{i-1} - t, t - 1)$; the first is calculated by p_i itself at previous steps, while the others are computed by p_{i-1} and, maybe, p_{i-2}.

Pebble $(kS_{i-1} - t, t)$ is the first one in row t to be calculated by processor p_i and it depends on pebbles $(kS_{i-1} - t - 1, t - 1)$, $(kS_{i-1} - t, t - 1)$ and $(kS_{i-1} - t + 1, t - 1)$; the last one has been computed by p_i itself at previous steps, while for the others two cases arise:

$ks_{i-1} > 1$: p_{i-1} computes the other two pebbles, that, by induction, are ready at time step $kt + D_{i-1}$ and will arrive to p_i at time step $kt + D_{i-1} + d_{i-1} = kt + D_i$. Thus, p_i computes $(kS_{i-1} - t, t)$ at time $kt + D_i + 1/s_i$.

$ks_{i-1} = 1$: p_{i-1} computes $(kS_{i-1} - t, t - 1)$ and p_{i-2} computes $(kS_{i-1} - t, t - 1) = (kS_{i-2} - t, t - 1)$ that, by induction, are ready at time steps $kt + D_{i-1}$ and $kt + D_{i-2}$, respectively. They will arrive at p_i at time step $kt + D_{i-1} + d_{i-1} = kt + D_{i-2} + d_{i-2} + d_{i-1} = kt + D_i$. Thus, p_i computes $(kS_{i-1} - t, t)$ at time $kt + D_i + 1/s_i$.

All the remaining pebbles of row t are computed by p_i with no more waiting in k steps. Thus pebble $(kS_i - 1 - t, t)$ is computed at time $kt + D_i + k \sum_{j=1}^{s_i} 1/s_i = k(t + 1) + D_i$. ■

Corollary 1 *The bottom half of triangle L can be calculated in $k(n/2) + D_{m-1}$ steps.*

Proof. The last processor to finish its computation is p_{m-1} with pebble $(n/2, n/2 - 1)$. ■

Theorem 1 *The slowdown of the Stripe algorithm is*

$$O\left(\min_I \left\{1 + \frac{n}{S_I} + \frac{m_I d_I}{n}\right\}\right),$$

where m_I *is the number of processors in* I*,* $S_I = \sum_{p_j \in I} s_j$ *is the total computation power of interval* I *and* $d_I = \left(\sum_{j=0}^{|I|-2} d_j\right)/m_I$ *is the average delay on the links between processors in interval* I*.*

Proof. Corollary 1 gives us the time needed to compute L's bottom half; the same time is needed to compute R's bottom half, while at most D_{m-1} steps are sufficient in order to exchange the necessary information to start the algorithm again on the next $n/2$ steps. Thus, the slowdown s associated with $n/2$ steps of computation, is upper bounded by

$$\begin{aligned} s &\leq \frac{2\left(kn/2 + D_{m_I-1}\right) + D_{m_I-1}}{n/2} \\ &= 2k + 6\frac{D_{m_I-1}}{n} \\ &\leq 2 + 2\frac{n}{S_I} + 6\frac{m_I D_{m_I-1}}{m_I n} \\ &\leq 2 + 2\frac{n}{S_I} + 6\frac{m_I d_I}{n}. \end{aligned}$$

■

2.1.2 Applications

The previous theorem gives us an upper bound on the slowdown as the minimum over all possible intervals I of a function of the speed, the number of processors and of the delay of the links connecting them. We now derive bounds for special cases in two different settings: first, processors in the host array all have the same speed and are more powerful that processors in the guest array; second, processors in the host array need not have the same speed.

Host with homogeneous processors. If processors in H are homogeneous and their speed is $S = S_m/m$, we distinguish the following cases:

1. $n \leq \sqrt{S d_{ave}}$:

   only processor p_1 is used to perform the simulation. p_1 needs at least one unit of time and at most n/S units of time to simulate one step of computation of G, thus

$$s \in O\left(1 + \sqrt{\frac{d_{ave}}{S}}\right).$$

2. $n > \sqrt{S d_{ave}}$:

   (a) If $n \geq S_m \geq m \geq \frac{n}{\sqrt{S d_{ave}}}$, then there must exist an interval $I \subseteq H$ of processors such that $m_I = \frac{n}{\sqrt{S d_{ave}}}$ and $d_I \leq d_{ave}$. (Suppose that such an interval does not exist and divide array H into n/m_I consecutive intervals of m_I processors each; from the fact that the average delay of every such interval must be greater than the average delay of the whole array, we can derive a contradiction.) Thus,

$$S_I = m_I S = n\sqrt{\frac{S}{d_{ave}}}$$

   and

$$s \in O\left(1 + \sqrt{\frac{d_{ave}}{S}}\right).$$

(b) If $n \geq S_m$ and $\frac{n}{\sqrt{S d_{ave}}} > m \geq 1$, the whole array H or a single processor is used to carry out the simulation and we have that

$$s \in O\left(\min\left\{\frac{n}{S_m} + \sqrt{\frac{d_{ave}}{S}}, \frac{n}{S}\right\}\right).$$

Thus, slowdown s can be as good as before when $S_m \in O(n)$, but can be $O(n)$ in very bad situations, *i.e.*, when the guest array has few ($m \in O(1)$) and not very powerful processors ($S_m \in O(1)$).

Host with heterogeneous processors. If processors in H are heterogeneous, with speed s_i for processor P_i, let $S_{ave} = S_m/m$ be the average speed of processors in H.

If we can find an interval $I \subseteq H$ such that

$$S_I m_I d_I = n^2 \text{ and } \frac{d_I}{S_{I_{ave}}} \leq \frac{d_{ave}}{S_{ave}},$$

where $S_{I_{ave}} = S_I/m_I$ is the average power of processors in I, then

$$s \in O\left(1 + \sqrt{\frac{d_{ave}}{S_{ave}}}\right).$$

If such an interval does not exist we have the cases:

1. $n > \sqrt{d_{ave}}$:

   If $S_m \geq m \geq n/\sqrt{d_{ave}}$, there must exist an interval $I \subseteq H$ such that $m_I = n/\sqrt{d_{ave}}$ (thus $S_m \geq n/\sqrt{d_{ave}}$) and $d_I \leq d_{ave}$. We have that

   $$s \in O\left(\sqrt{d_{ave}}\right).$$

   The same slowdown can be achieved also if $S_m \geq n/\sqrt{d_{ave}} > m$ using the whole H for the simulation.

2. $n \leq \sqrt{d_{ave}}$:

   The processor with maximum speed s_{max} is used to perform the simulation. It needs at least one unit of time and at most n/s_{max} units of time to simulate one step of computation of G. As $s_{max} \geq S_m/m$, we have

   $$s \in O\left(1 + \frac{\sqrt{d_{ave}}}{S_{ave}}\right).$$

2.1.3 A lower bound

In this section we show that the upper bound $s \leq \min_I O(1 + \frac{n}{S_I} + \frac{m_I d_I}{n})$ is asymptotically tight.

Lemma 2 *Pebble* (i, n) *can not be computed earlier than time step*

$$\min_I \max\{n^2/2S_I, m_I d_I/2\}.$$

Proof. Consider any simulation that uses interval I of processors in H. There must exist a subinterval $I' = \{p_j, \ldots, p_{j+|I'|-1}\} \subseteq I$ of consecutive processors such that p_j and $p_{|I'|}$ are the two farthest apart processors that must exchange information during the simulation before pebble (i, n) is computed. Thus, (i, n) cannot be calculated in less than $m_{I'} d_{I'}/2$ time steps.

Moreover, in order to compute pebble (i, n) we first need to compute every pebble in triangle $((1, 1), (n, 1), (i, n))$, that needs at least $n^2/2S_{I'}$ time steps to be computed. ■

Corollary 2 *The first kn steps of computation can not be simulated in less than*

$$k \min_I \max\{n^2/2S_I, m_I d_I/2\}$$

time steps.

By the previous corollary we can, thus, state the following theorem:

Theorem 2 *The slowdown of the best simulation of G by H is*

$$\Theta\left(\min_I \{1 + n/S_I + m_I d_I/n\}\right).$$

2.2 Links without pipelining

We now analyse the case in which links between processors do not have the possibility to pipeline messages; *i.e.*, a new message can be sent on a link only when the preceding one has arrived at its destination. We describe the simulation only in the case of homogeneous processors and we show that the slowdown is upper bounded by the delay on the links and that the simulation is *work efficient*; i.e. we simulate m steps of computation of an n-processor unit-delay linear array with a n/d-processor linear array with links of delay d in time $O(md)$. The case of heterogeneous processors is a straightforward generalisation.

We are given a guest array G of n processors $p_i, i = 0, \ldots, n-1$, with unit delays on links, that computes a DAG $D' = \{(x, y) \mid x \leq n-1 \text{ and } 0 \leq y \leq m, m \geq n\}$ and a host array H of $p \leq n$ processors with delay $d > 1$ on links (w.l.o.g. we suppose n is a multiple of $d+1$). We also assume tht the links of H do not support pipelining.

DAG D' can be computed by H in a work-efficient way using $n/(d+1)$ processors and with a slowdown of $O(d)$ in the following way. The dag D' is divided into $n/(d+1)$ vertical stripes St_k each $d+1$ pebbles wide. More precisely, the computation goes as follows:

- set $St_k = \{(x, y) \mid 0 \leq y \leq m, k(d+1) \leq x < (k+1)(d+1)\}$, $k = 0, \ldots, n/(d+1)$;
- set $left(k, r) = (k(d+1), r)$ and $right(k, r) = (k(d+1)+d, r)$ for every $0 \leq r \leq m$ and every $0 \leq k \leq n/d+1$;
- processor p_k computes stripe St_k row by row in a bottom-up manner. Pebbles in row r are computed in the following order:

$$\begin{array}{ll} left(k,r), right(k,r), (k(d+1)+1, r), (k(d+1)+2, r), \ldots & \\ \qquad (k(d+1)+d-1, r) & \text{if } k \bmod 2 = 0 \\ right(k,r), left(k,r), (k(d+1)+1, r), (k(d+1)+2, r), \ldots & \\ \qquad (k(d+1)+d-1, r) & \text{if } k \bmod 2 = 1 \end{array}$$

- All processors start computation at $t = 0$.

We now prove that every processors has at its disposal all the pebbles needed to carry out its computation and that the computation of H has a slowdown of $O(d)$. We start by defining $t(x, y)$ as the time by which pebble (x, y) is computed, and by $prev(x, y)$ as the number of pebbles that are computed before pebble (x, y) by the same processor. We have that

Lemma 3 *For every $0 \leq r \leq m$, $t(x, r) = prev(x, r) + 1$.*

Proof. The claim clearly holds for $r = 0$. Suppose it holds for fixed $r \geq 0$. If we prove that, for every k, the claim holds for $right(k, r+1)$ and $left(k, r+1)$ then it holds also for the remaining pebbles in row $r+1$. In fact, once p_k has computed $right(k, r+1)$ and $left(k, r+1)$, then it can compute the other pebbles in row r of its stripe one at each step with no more waiting. We prove the claim only for $right(k, r+1)$; the proof for $left(k, r+1)$ is analogous.

Notice that

$$t(right(k, r+1)) = \max\{t(right(k, r)) + d + 1, t(left(k-1, r)) + d + 1\};$$

that is, p_k can compute $right(k, r+1)$ when it has computed all the pebbles of the previous row and when $left(k-1, r)$, computed by p_{k-1}, has arrived.

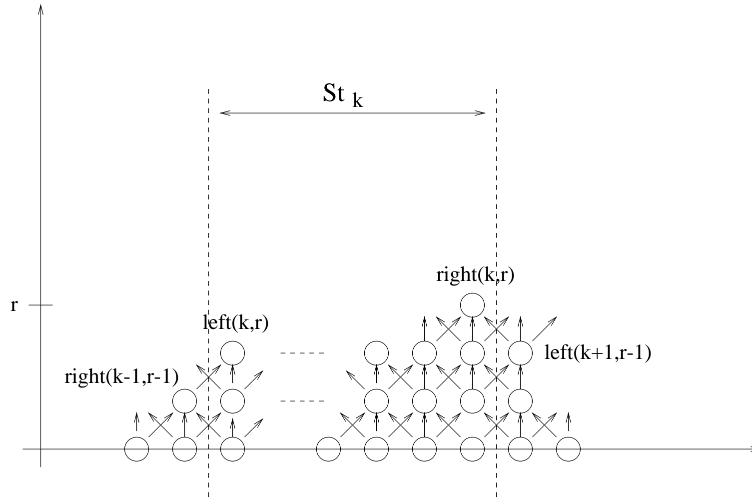

Figure 2: Processor p_k, $k = 0, \ldots n/(d+1)$, computes a vertical stripe of width $d+1$ pebbles, from $(k(d+1), 0)$ to $((k+1)(d+1), 0)$. $left(k,r)$ is the leftmost pebble of row r in stripe St_k, while $right(k,r)$ is the rightmost of the same row and same stripe.

Using the inductive hypothesis we have

$$t(right(k,r)) + d + 1 = prev(right(k,r)) + 1 + d + 1 = prev(right(k, r+1)) + 1.$$

As, for every $r \in [0, m]$ and for every $k \in [0, n/d+1)$, $prev(left(k,r)) = prev(right(k+1, r))$, using the inductive hypothesis

$$\begin{aligned} t(left(k-1,r)) + d + 1 &= prev(left(k-1,r)) + 1 + d + 1 \\ &= prev(right(k,r)) + d + 1 + 1 \\ &= prev(right(k, r+1)) + 1. \end{aligned}$$

In conclusion:

$$t(right(k, r+1)) = prev(right(k, r+1)) + 1$$

■

Corollary 3 *The computation of stripe St_k is finished by time $t = m(d+1)$.*

Proof. For every k, $(k(d+1)+d-1, m)$ is the last pebble to be computed in stripe St_k and, because of Lemma (3), $t(k(d+1)+d-1, m) = prev(k(d+1)+d-1, m) + 1 = |S_k| = m(d+1)$. ■

Theorem 3 *The simulation above is work-efficient.*

Proof. We use $n/(d+1)$ processors to simulate m steps of computation of n processors in $m(d+1)$ time. ■

3 Simulating cliques

In this section we present an automatic method for simulating in the database model (see [2]) homogeneous processor cliques with unit delay links, on both homogeneous and heterogeneous cliques of processors with arbitrary delays on links that disallow pipelining. In the database model, each processor p has a potentially large local database that may be accessed only by p at each step of computation. Before the simulation starts it is possible to assign the databases of the guest machine to the processors of the host machine. However, the size of the database makes it infeasible for two processors to exchange databases once the simulation has started and only updates of the database can be passed.

These results have straightforward implications for simulating PRAM algorithms on an arbitrary NOW. A *shared-memory PRAM* is an abstract model of parallelism which consists of n processors and a global shared memory of size M. Each processor has its own local control and its own local memory. During each step of a shared-memory PRAM computation, each processor is allowed to access any location of the global shared memory and to perform some computation according to its local control and its local memory. Here we consider a variation of the shared-memory PRAM model called the *distributed-memory PRAM* (also called a distributed memory machine or DMM). Here, the memory of size M is distributed evenly among the n processors with each processor receiving a block of memory of M/n locations. In the distributed memory PRAM, each processor has direct access to its own memory and to every other processor but it has only indirect access to other processors' memory. Moreover, at each time step, each memory block can be accessed by at most one processor. Using well-known techniques related to random hashing, a shared-memory PRAM can be simulated by a distributed-memory PRAM with a slowdown of $O(\log n)$ with high probability (see [12] for references). By seeing the guest clique as a distributed memory PRAM and the host clique to a NOW with the weight $d_{i,j}$ of edge (i, j) representing the delay of the minimum-delay path in the NOW from vertex processor p_i to processor p_j, we can get a method for automatically compiling PRAM algorithms for NOWs.

3.1 Homogeneous processors

In this section, we are given an n-vertex weighted host clique C and we use it to simulate an n-vertex unit delay clique. We show that the slowdown is proportional to the average of the delays of the links represented by weights.

Given a weighted clique $C = (V, E)$, with n vertices and weight d_e on edge $e \in E$ we define the subgraph $H = (V, E')$ such that:

$$E' = \{e \in E \text{ such that } d_e \leq d_{ave}\}$$

where d_{ave} is the average weight on C_n's edges. A node is said to be *alive* if it has degree at least $n/2$ in H; otherwise it is *dead*.

The simulation works in the following way: distribute the databases of the guest processors equally among the alive nodes in H; use both alive and dead nodes in H for message passing; have each alive node perform all the computation related to its assigned processors.

Lemma 4 *Any two alive nodes are either adjacent or share a common (dead or alive) neighbor in H.*

Proof. Suppose by contradiction that there exist two nodes u_1 and u_2 that are not adjacent and such that the intersection of their neighbor sets V_1 and V_2 is empty. As both u_1 and u_2 are alive, $|V_1|, |V_2| \geq n/2$. Thus, $|V_1 \cup V_2| \geq n$, but this is a contradiction since $u_1 \notin V_2$, $u_2 \notin V_1$. ■

Lemma 5 *The alive nodes are a constant fraction of n.*

Proof. The maximum number of eliminated edges is $\binom{n}{2}/2$, thus the maximum number of dead nodes is $2\binom{n}{2}/2n = (n-1)/2$. Therefore at least $n - (n-1)/2 > n/2$ are alive. ■

Theorem 4 *An n-vertex clique with links without pipelining and with average delay d_{ave} can simulate an n-vertex clique with a slowdown $O(d_{ave})$.*

Proof. Since the number of alive nodes is a constant fraction of the total number of nodes, it is possible to assign guest processors (along with their local database) to host processors so that each host is responsible for a constant number of processors. As the distance between every pair of alive nodes is at most 2 and the delay of every used link is at most d_{ave}, the time spent for communicating at each step is at most $O(d_{ave})$. ■

It is easy to see that in a NOW in which all links have the same delay $d = O(n)$ no simulation can achieve a slowdown smaller than d; therefore our simulation is asymptotically optimal. Conversely, if $d = \Omega(n)$, the trivial simulation that assigns work only to one processor of the NOW achieves a slowdown of $O(n)$.

3.2 Heterogeneous Processors

In this section we briefly discuss how to extend the simulation of the previous section to the case in which the host network is a clique of m heterogeneous processors. The basic idea is to expand a vertex of the clique

corresponding to a processor with computing power s into a clique of s processors connected among themselves with links of delay 0 and then use the simulation with $O(d_{ave})$ slowdown of the previous section using this new graph as host.

As before, the host consists of processors $p_0, \cdots, p_{m-1}$ and is represented by a complete graph $C = (V, E)$ on m vertices that has weights on the vertices and on the nodes. The weight $d_{i,j}$ of edge (i, j) represents the delay on the edge (i, j) and weight s_i of node i represents the speed of processor p_i. We denote with S_I, for $I \subseteq \{0, \cdots m-1\}$, the sum of the speeds of the processors of I and by S the sum of the speeds of all processors. Weight s_i of node i represents the speed of processor p_i. We assume that weights on both edges and nodes are integers and that $S \leq n$.

We start by defining a graph G' with unweighted nodes; on this graph we will perform the simulation presented in the previous section. We then observe that C can simulate G' without any additional slowdown.

Let $G' = (V', E')$ be the edge-weighted clique defined as follows:

- $V' = \{v_{i,j} \mid 0 \leq i \leq n-1, 0 \leq j \leq s_i - 1\}$ (thus $|V'| = S$);
- weight $d(v_{i,j}, v_{l,k})$ on edge $(v_{i,j}, v_{l,k})$ is defined in the following way:

$$d(v_{i,j}, v_{l,k}) = \begin{cases} 0 & \text{if } i = l \\ d_{i,l} & \text{otherwise} \end{cases}$$

G' can perform the simulation described in the previous section achieving slowdown s'

$$s' \in O\left(\frac{\sum_{e \in E'} d(e)}{|sE'|}\right) = O\left(\frac{\sum_{i,j=0\ i \neq j}^{n-1} d_{i,j} s_i s_j}{S(S-1)}\right).$$

Now, C simulates G' in the following way: every processor p_i, $i = 0, \ldots, n-1$, performs the computation of all processors in $V_i = \{v_{i,j} \in V' \mid 0 \leq j \leq s_i - 1\}$ with constant slowdown. Therefore the slowdown of the simulation of the host network by C is $O(s')$.

4 Open problems

In this paper we presented simulations of linear arrays on linear arrays and of cliques on cliques. For example, our simulation of a clique by a clique guarantees a slowdown proportional to the average host link delay. Besides designing simulations between other pairs of important DAGs, it would be interesting to design algorithms that map guest processors on host processors so to guarantee optimal simulation. Alternatively, one could give evidence of the hardness of the problem and present approximate algorithms.

References

[1] M. Andrews, T. Leighton, P.T. Metaxas, L. Zhang. *Automatic Method for Hiding Latency in High Bandwidth Networks.* In Proc. of the ACM Symposium on Theory of Computing, 1996.

[2] M. Andrews, T. Leighton, P.T. Metaxas, L. Zhang. *Improved Methods for Hiding Latency in High Bandwidth Networks.* In Proc. of the 8th annual ACM Symposium on Parallel Algorithms and Architectures, pages 52-61.

[3] Y. Aumann, M. Ben-Or. *Computing with Faulty Arrays.* In Proc. of the 24th Annual ACM Symposium on Theory of Computing, pp. 162- 169, 1992.

[4] R. Cole, B. Maggs, R. Sitaraman. *Multi-scale Self-simulation: a Technique for Reconfiguring Arrays with Faults.* In Proc. of the 25th Annual ACM Symposium on Theory of Computing, pp.561-572, 1993.

[5] C. Kaklamanis, A.R. Karlin, F.T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, A. Tsantilas. *Asymptotically Tight Bounds for Computing with Faulty Array of Processors.* In Proc of the 31st Annual Symposium on Foundation of Computer Science, pp. 285-296, 1990.

[6] R. Koch, T. Leighton, B. Maggs, S. Rao, A. Rosenberg. *Work-Preserving Emulations of Fixed-Connection Networks.* In Proc. of the 21st Annual ACM Symposium on Theory of Computing, pp.227-240, 1990.

[7] T. Leighton, B. Maggs, R. Sitaraman. *On the Fault Tolerance of Some Popular Bounded Degree Networks.* In Proc. of the 33rd Annual Symposium on Foundation of Computer Science, pp.542-552, 1992.

[8] C.E. Leiserson, S. Rao, S. Toledo. *Efficient Out-of-Core Algorithms for Linear Relaxation Using Blocking Covers.* In Proc. of the 34th Annual Symposium on Foundation of Computer Science, pp. 704-713, 1993.

[9] M.O. Rabin. *Efficient dispersal of Information for Security, Load Balancing and Faults Tolerance.* Journal of the ACM, 36(2), pp. 335- 348, 1989.

[10] L.G. Valiant. *Bulk-synchronous Parallel Computers.* Technical Report TR-08-89, Center of Research in Computing Technology, Harvard University, 1989.

[11] L.G. Valiant. *A Bridging Model for Parallel Computation.* Communication of the ACM, 33(8), pp. 103-111, 1990.

[12] L.G. Valiant. *General Purpose Parallel Architectures* in *Handbook of Theoretical Computer Science*, J. van Leeuwen (editor). Elsevier, Amsterdam, 1990.