

# Efficient automatic simulation of parallel computation on networks of workstations

Christos Kaklamanis<sup>a,1,2</sup>, Danny Krizanc<sup>b</sup>, Manuela Montangero<sup>c,\*</sup>,  
Giuseppe Persiano<sup>d,2</sup>

<sup>a</sup>Computer Technology Institute and Department of Computer Engineering and Informatics, University of Patras, GR26500 Rion, Greece

<sup>b</sup>Department of Mathematics and Computer Science, Wesleyan University, Middletown CT 06459, USA

<sup>c</sup>Dipartimento di Ingegneria dell'Informazione, Università di Modena e Reggio Emilia, Via Vignolese 905/b, 41100 Modena, Italy

<sup>d</sup>Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (Salerno), Italy

Received 14 October 2003; received in revised form 30 May 2005; accepted 29 October 2005

Available online 23 March 2006

## Abstract

Andrews et al. [Automatic method for hiding latency in high bandwidth networks, in: Proceedings of the ACM Symposium on Theory of Computing, 1996, pp. 257–265; Improved methods for hiding latency in high bandwidth networks, in: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, 1996, pp. 52–61] introduced a number of techniques for automatically hiding latency when performing simulations of networks with unit delay links on networks with arbitrary unequal delay links. 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 connected by unit-delay links on a linear array of heterogeneous processors connected by links with arbitrary delay. We show that the slowdown achieved by our simulation is optimal. We then consider the case of simulating cliques by cliques; i.e., a clique of heterogeneous processors with arbitrary delay links is used to simulate a clique of homogeneous processors with unit delay links. We reduce the slowdown from the obvious bound of the maximum delay link to the average of the link delays. In 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.

© 2006 Elsevier B.V. All rights reserved.

**Keywords:** Parallel computation; Distributed computation; Automatic simulation

\* Corresponding author.

E-mail addresses: [kakl@cti.gr](mailto:kakl@cti.gr) (C. Kaklamanis), [dkrizanc@wesleyan.edu](mailto:dkrizanc@wesleyan.edu) (D. Krizanc), [montangero.manuela@unimo.it](mailto:montangero.manuela@unimo.it) (M. Montangero), [giuper@dia.unisa.it](mailto:giuper@dia.unisa.it) (G. Persiano).

<sup>1</sup> Work partially supported by the European RTN Project under contract HPRN-CT-2002-00278, COMBSTRU.

<sup>2</sup> Work partially supported by the European Integrated Project under contract FP6-015964, AEOLUS.

## 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)). A NOW is a very attractive and widely available type of distributed systems 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, during week-ends, lunch breaks, and meetings) 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 [3,5,6,8–12]. 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 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. [1,2] consider 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.

Andrews et al. concentrate their attention on parallel algorithms for linear arrays automatically simulated by 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  $d_0, \dots, d_{n-1} > 1$  on its links and average delay  $d_{\text{ave}}$ . 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 steps 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 the preceding step. 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. [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  $O(\sqrt{d_{\text{ave}}})$ . In [2] they show that in the database model a slowdown of  $O(\sqrt{d_{\text{ave}}} \cdot \log^3 n)$  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 queueing 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 cases 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 analyze both the cases in which processors are homogeneous and heterogeneous. 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. Allowing compilation of PRAMs 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. This has the potential to shortcut parallel software development cycles considerably.

## 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  $q_i$ ,  $i = 0, \dots, n - 1$ . Processor  $q_i$ ,  $0 < i < n - 1$ , can communicate with its two neighbors  $q_{i-1}$  and  $q_{i+1}$ , while processor  $q_0$  can communicate only with  $q_1$  and  $q_{n-1}$  only with  $q_{n-2}$ . 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  $q_x$  at time step  $y$ . The computation  $(x, y)$  depends on the outcome of the computation of  $q_x$  and its neighbors 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 processors  $p_i$ ,  $i = 0, \dots, m - 1$  with  $m \leq n$ , 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 to 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 computation 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 link 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. Algorithm Stripes

Consider the first  $n$  steps of  $G$ 's computation. Define  $L$  as the triangle formed by the vertices  $(j, t)$  of the DAG such that  $j \leq n - t$  and  $R$  as the one formed by vertices  $(j, t)$  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 vertex of the first  $n/2$  steps are simulated. In the same way it will simulate the next steps of computation,  $n/2$  by  $n/2$ .

Only a portion of array  $H$  is used to perform the simulation; for the sake of presentation and w.l.o.g. we suppose that we use processors in the interval  $I = \{p_0, \dots, p_{m_I-1}\} \subseteq H$ , where  $m_I \leq m$ .

We will use the following notation:

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

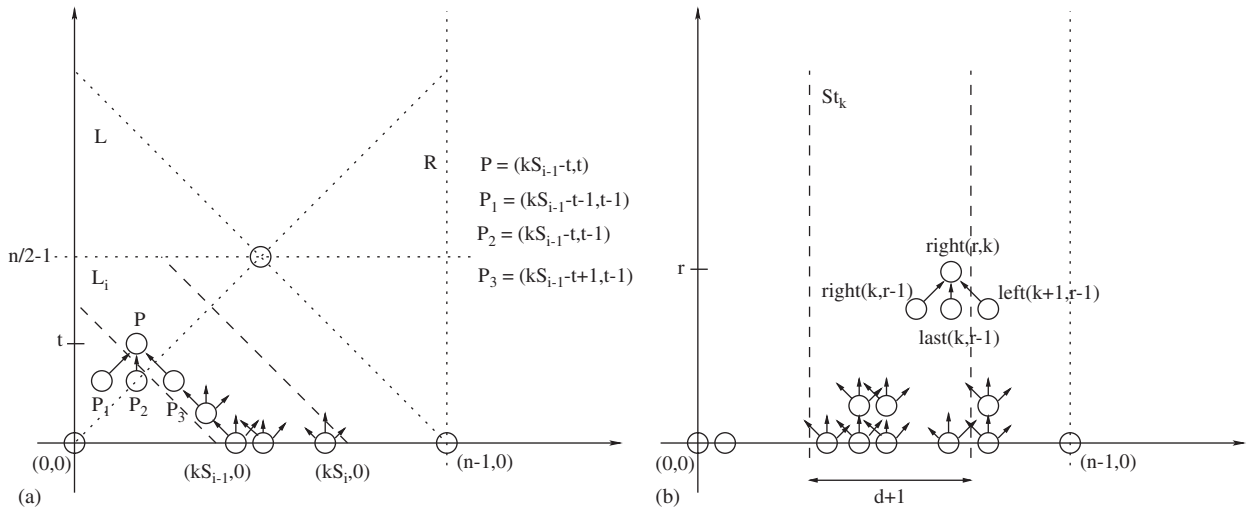


Fig. 1. (a) Simulation in the case links allow pipeline. (b) Simulation in the case links do not allow pipeline.

Let  $k = \lceil n/S_I \rceil$ . We divide the bottom half of  $L$  into slanting stripes and each processor  $p_i$  computes the stripe  $L_i$  of width  $ks_i$  defined in the following way (see Fig. 1a):

$$L_i = \{(j, t) | t = 0, \dots, T_i \text{ and } j = J_i, \dots, ks_i - 1 - t\},$$

where  $T_i = \min\{n/2 - 1, ks_{i-1}\}$  and  $J_i = \max\{0, ks_{i-1} - t\}$ .

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 vertex  $(ks_i - t, t)$  at time step less or equal to  $k(t + 1) + D_i$ .

**Proof.** First observe that the computation of processor  $p_i$  depends on vertices 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 neighbors, it could compute all the vertices on any row  $t$  of its stripe 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, vertex  $(ks_0, 0)$ , 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.

Vertex  $P = (ks_{i-1} - t, t)$  is the first one in row  $t$  to be calculated by processor  $p_i$  and it depends on vertices  $P_1 = (ks_{i-1} - t - 1, t - 1)$ ,  $P_2 = (ks_{i-1} - t, t - 1)$  and  $P_3 = (ks_{i-1} - t + 1, t - 1)$  (see Fig. 1(a)); vertex  $P_3$  has been computed by  $p_i$  itself at previous steps, while for the others two cases might arise:

$ks_{i-1} > 1$ :  $p_{i-1}$  computes  $P_1$  and  $P_2$ , that, by induction, are ready at time step  $kt + D_{i-1}$  and will arrive to  $p_i$  at time step  $kt + D_{i-1} + d_{i-1} = kt + D_i$ .

$ks_{i-1} = 1$ :  $p_{i-1}$  computes  $P_2$  and  $p_{i-2}$  computes  $P_1$  that, by induction, are ready, respectively, at time steps  $kt + D_{i-1}$  and  $kt + D_{i-2}$ . They will arrive to  $p_i$  at time step  $kt + D_{i-1} + d_{i-1} = kt + D_{i-2} + d_{i-2} + d_{i-1} = kt + D_i$ .

Thus,  $P_1, P_2$  and  $P_3$  are available for  $p_i$  at time  $kt + D_i$  and the computation of row  $t$  can be computed in  $k$  steps by time  $kt + D_i + k = k(t + 1) + D_i$ .  $\square$

**Corollary 1.** The bottom half of triangle  $L$  can be calculated in  $kn/2 + D_I$  time steps.

**Proof.** The last processor to finish its computation is  $p_{m_I-1}$  with vertex  $(n/2 - 1, n/2 - 1)$ .  $\square$

**Theorem 1.** Algorithm Stripes has slowdown

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

of host time steps per guest time steps, where  $m_I$  is the number of processors in  $I$ ,  $S_I$  is the total computation power of interval  $I$  and  $d_I = D_I/m_I$  is the average delay on the links between processors in interval  $I$ .

**Proof.** By Corollary 1 we know how much time is needed to compute  $L$ 's bottom half; the same time is needed to compute  $R$ 's bottom half and at most  $D_I$  time steps to exchange the necessary information to start the algorithm again on the next  $n/2$  steps. This because, in general, processors will start the computation on the next  $n/2$  steps from vertices they have not computed in the previous ones; e.g., vertex  $(n/2 - 1, n/2 - 1)$  is computed by processor  $p_{m_I-1}$  at step  $n/2 - 1$ , but vertex  $(n/2 - 1, n/2)$  will be computed by processor  $p_j$ , with  $j \leq m_I - 1$ , at step  $n/2$ .

Thus, slowdown  $\ell$ , computed on  $n/2$  steps of computation, is upper bounded by the time needed to compute the  $n/2$  steps divided by the number of steps.

$$\begin{aligned} \ell &\leq \frac{2(kn/2 + D_I) + D_I}{n/2} \\ &= 2k + 6\frac{D_I}{n} \\ &\leq 2 + 2\frac{n}{S_I} + 6\frac{m_I D_I}{m_I n} \quad \text{as } k \leq 1 + n/S_I \\ &\leq 2 + 2\frac{n}{S_I} + 6\frac{m_I d_I}{n}. \quad \square \end{aligned}$$

2.1.2. Discussion

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 and 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 than processors in the guest array; second, processors in the host array do not necessarily have the same speed. We denote with  $d_{ave} = D_m/(m - 1)$  the average delay on links in  $H$ .

*Homogeneous processors:* Let  $s$  be the speed of processors in  $H$ , then the total computation power of  $H$  is given by  $S_m = s \cdot m$ . We distinguish the following cases:

(1)  $n \leq \sqrt{s \cdot d_{ave}}$ : only one processor is used to perform the simulation. This processor needs at least one unit of time and at most  $\lceil n/s \rceil = O(1 + n/s)$  units of time to simulate one step of computation of  $G$ , thus we have

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

(2)  $n > \sqrt{s \cdot d_{ave}}$ : we further distinguish the following cases:

(a) If  $n \geq S_m \geq m \geq n/\sqrt{s \cdot d_{ave}}$  then there must exist an interval  $I \subseteq H$  of consecutive processors such that  $m_I = n/\sqrt{s \cdot d_{ave}}$  and  $d_I \leq d_{ave}$ . Suppose by contradiction that such an interval does not exist; divide the array  $H$  into  $h = (m - 1)/(m_I - 1)$  consecutive intervals of  $m_I$  processors each, such that intervals share endpoints. As we are interested in asymptotic analysis we assume w.l.o.g. that  $m_I - 1$  divides  $m - 1$ . Let  $\Delta_i > d_{ave}$  be the average delay on links of the  $i$ th interval, thus

$$d_{ave} = \frac{\sum_{i=1}^h (m_I - 1)\Delta_i}{m - 1} = \frac{(m_I - 1)}{m - 1} \sum_{i=1}^h \Delta_i > \frac{m_I - 1}{m - 1} \frac{m - 1}{m_I - 1} d_{ave},$$

which is a contradiction. Thus, given  $I$ , we have

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

and by (1)

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

(b) If  $n \geq S_m$  and  $n/\sqrt{s \cdot d_{ave}} > m \geq 1$ , the whole array  $H$  or a single processor is used to carry out the simulation, according to which solution performs better. From (1) we have that

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

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

*Heterogeneous processors:* Let  $s_i$  be the speed of processor  $p_i$  and  $S_{ave} = S_m/m$  be the average speed of processors in  $H$ .

If there exists an interval  $I \subseteq H$  such that

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

where  $S_{I_{ave}} = S_I/m_I$  is the average power of processors in  $I$ , we use interval  $I$  to perform the simulation. Referring to (1) we have that  $n/S_I = m_I d_I/n$  and

$$\frac{m_I d_I}{n} = \frac{m_I d_I}{\sqrt{S_I m_I d_I}} = \sqrt{\frac{m_I d_I}{S_I}} = \sqrt{\frac{d_I}{S_{I_{ave}}}} \leq \sqrt{\frac{d_{ave}}{S_{ave}}},$$

hence

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

If such an interval does not exist, one of the following cases must arise:

(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  $d_I \leq d_{ave}$  (the argument of the existence of interval  $I$  is analogous to the one in 2(a)). Using  $I$  to perform the simulation by (1) we have

$$\ell \in O(\sqrt{d_{ave}}).$$

The same slowdown is achieved when  $S_m \geq n/\sqrt{d_{ave}} > m$  using the whole array  $H$  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  $s_{max} \geq S_m/m$ , from (1) we have

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

### 2.1.3. A lower bound

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

**Lemma 2.** Vertex  $(i, n)$  cannot be computed earlier than time step

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

**Proof.** Consider any simulation that uses an interval  $I$  of processors in  $H$ . There must exist a subinterval  $I' = \{p_j, \dots, p_{j+|I'|-1}\} \subseteq I$  of consecutive processors such that  $p_j$  and  $p_{|I'|}$  are the farthest processors that must exchange information during the simulation before vertex  $(i, n)$  is computed. Thus,  $(i, n)$  cannot be calculated in less than  $m_{I'} d_{I'}/2$  time steps. Moreover, to calculate vertex  $(i, n)$  we first need to calculate every vertex in triangle  $((1, 1), (n, 1), (i, n))$ , that needs at least  $n^2/2S_{I'}$  time steps to be computed.  $\square$

**Corollary 2.** *The first  $kn$  steps of computation cannot 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*

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

*time steps.*

## 2.2. Links without pipelining

We now analyze 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 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  $O(md)$ ; i.e., the simulation is *work efficient*. We describe the simulation in detail for the case processors are homogeneous. The case of heterogeneous processors is a straightforward generalization.

We are given a guest array  $G$  of  $n$  processors  $q_i$ ,  $i = 0, \dots, n-1$ , 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  $h \leq n$  processors  $p_i$  with delay  $d > 1$  on links not supporting pipelining. W.l.o.g. assume  $n$  is a multiple of  $d+1$ .

DAG  $D'$  can be computed by  $H$  in a work-efficient way using  $n/(d+1)$  processors with a slowdown of  $O(d)$ : divide  $D'$  into  $n/(d+1)$  vertical stripes  $St_k$  each  $d+1$  vertices wide and assign each stripe to one processor of the host array.

More precisely, the computation goes as follows:

- Set  $St_k = \{(x, y) \mid 0 \leq y \leq m, k(d+1) \leq x < (k+1)(d+1)\}$ ,  $k = 0, \dots, n/(d+1)$ ; i.e.,  $St_k$  is a vertical stripe of dag  $D'$  of width  $d+1$  vertices.
- Set  $left(k, r) = (k(d+1), r)$ ,  $right(k, r) = (k(d+1) + d, r)$  and  $last(k, r) = (k(d+1) + d - 1, r)$  for every  $0 \leq r \leq m$  and every  $0 \leq k \leq n/d + 1$ ; i.e.,  $left(k, r)$  is the leftmost vertex of stripe  $St_k$  at row  $r$  and, analogously,  $right(k, r)$  is the rightmost of the same stripe at the same row.  $last(k, r)$  is the vertex on the left of  $right(k, r)$  and will be the last one to be computed in row  $r$ .
- Processor  $p_k$  computes stripe  $St_k$  row by row in a bottom-up manner. Vertices in row  $r$  are computed in the following order:

if  $k \bmod 2 = 0$  then  
 $left(k, r), right(k, r), (k(d+1) + 1, r), (k(d+1) + 2, r), \dots, last(k, r)$   
 if  $k \bmod 2 = 1$  then  
 $right(k, r), left(k, r), (k(d+1) + 1, r), (k(d+1) + 2, r), \dots, last(k, r),$

i.e., the first vertices to be computed in every row are the leftmost and the rightmost and the order in which this is done depends on the position of the stripe. All the other vertices in the row are computed from left to right.

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

We now prove that every processors has always at its disposal all the vertices needed to carry out its computation, i.e., it can compute a vertex at every instant of time, and that the slowdown of the computation of  $H$  is  $O(d)$ . We start by defining  $t(x, y)$  as the time by which vertex  $(x, y)$  is computed, and by  $prev(x, y)$  as the number of vertices that are computed before vertex  $(x, y)$  by the same processor.

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

**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  $\text{right}(k, r + 1)$  and  $\text{left}(k, r + 1)$  then it holds also for the remaining vertices in row  $r + 1$ . In fact, except for  $\text{right}(k, r + 1)$  and  $\text{left}(k, r + 1)$ , the processor itself computes all the vertices needed for vertices in row  $r + 1$ . We prove the claim only for  $\text{right}(k, r + 1)$  and  $k$  odd, the proof for  $\text{left}(k, r + 1)$  and the cases  $k$  even are analogous.

To compute  $\text{right}(k, r + 1)$ , processor  $p_k$  needs to have the information relative to vertices  $\text{last}(k, r)$ ,  $\text{right}(k, r)$ ,  $\text{left}(k + 1, r)$  (see Fig. 1(b)). As  $\text{right}(k, r)$  is computed earlier than  $\text{last}(k, r)$  then

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

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

Using the inductive hypothesis we have

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

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

**Proof.** For every  $k$ ,  $\text{last}(k, m)$  is the last vertex to be computed in stripe  $St_k$  and, because of Lemma 3,  $t(\text{last}(k, m)) = |S_k| = m(d + 1)$ .  $\square$

**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.  $\square$

When processors are heterogeneous the simulation works in the same way with the only difference that stripe  $St_k$ , computed by processor  $p_k$  with speed  $s_k$ , has width  $s_k(d + 1)$ .

### 3. Simulating cliques

In this section we present an automatic method for simulating, in the database model, 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 equating the guest clique to 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 get a method of automatically compiling PRAM algorithms for NOWs.

### 3.1. Homogeneous processors

In this section, we are given an  $n$ -vertex host clique  $C$  and use it to simulate an  $n$ -vertex unit delay clique and show that the slowdown is order of the average of the weights on links.

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

$$E' = \{e \in E \text{ such that } d_e \leq 2d_{\text{ave}}\},$$

where  $d_{\text{ave}}$  is the average weight edges in  $E$ . 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: equally distribute the databases of the guest processors among the alive nodes in  $H$  and use alive and dead nodes for message passing. Each alive node will perform all the computation relative to the processors assigned to it.

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

**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,  $|V_1|, |V_2| \geq n/2$ . As  $V_1 \cap V_2 = \emptyset$  then  $|V_1 \cup V_2| \geq n$ , but this is a contradiction since  $u_1 \notin V_2, u_2 \notin V_1$ .  $\square$

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

**Proof.** First note that the number of eliminated edges is at most  $\binom{n}{2} / 2$ . Suppose by contradiction that  $|E - E'| > \binom{n}{2} / 2$ , then

$$d_{\text{ave}} = \frac{\sum_{e \in E - E'} d_e + \sum_{e \in E'} d_e}{\binom{n}{2}} > \frac{\left(\binom{n}{2} / 2\right) 2d_{\text{ave}}}{\binom{n}{2}} = d_{\text{ave}}.$$

Thus, the number of dead nodes is at most  $2 \binom{n}{2} / 2n = (n - 1) / 2$  and at least  $n - (n - 1) / 2 > n / 2$  are alive.  $\square$

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

**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  $2d_{\text{ave}}$ , the time spent for communicating at each step is at most  $O(d_{\text{ave}})$ .  $\square$

It is easy to see that in a NOW in which all links have the same delay  $d = O(n)$ , then no simulation can achieve a slowdown smaller than  $d$  and thus 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 the extension of 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 computer power  $s$  into a clique of  $s$  processors connected among themselves with links of delay 0 and then use the simulation with  $O(d_{\text{ave}})$  slowdown using this new graph as host.

As before, the host consists of processors  $p_0, \dots, p_{m-1}$  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 and the weight  $s_i$  on node  $i$  represents the speed of processor  $p_i$ . We denote by  $S$  the sum of the speeds of all processors. We assume that weights on both edges and nodes are integer and that  $1 \leq S \leq n$ .

We start by defining a graph  $G'$  with unweighted nodes; using this graph we can define a simulation with the technique given 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  $\ell'$

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

Now,  $C$  simulates  $G'$  in the following way: every processor  $p_i$ ,  $i = 0, \dots, n-1$ , performs the computation of all processors in  $V_i = \{v_{i,j} \in V' \mid 0 \leq j \leq s_i-1\}$ . The slowdown of the simulation using  $C$  is still  $\ell'$ .

#### 4. Discussion and open problems

Our simulation of a clique by a clique guarantees a slowdown proportional to the average delay. It would be interesting to design algorithms that map guest processors to host processors so as to guarantee optimal simulation or to 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: Proceedings of the ACM Symposium on Theory of Computing, 1996, pp. 257–265.
- [2] M. Andrews, T. Leighton, P.T. Metaxas, L. Zhang, Improved methods for hiding latency in high bandwidth networks, in: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, 1996, pp. 52–61.
- [3] Y. Aumann, M. Ben-Or, Computing with faulty arrays, in: Proceedings of the 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 162–169.
- [4] R. Cole, B. Maggs, 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, 1993, pp. 561–572.
- [5] C. Kaklamani, 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: Proceedings of the 31st Annual Symposium on Foundation of Computer Science, 1990, pp. 285–296.
- [6] R. Koch, T. Leighton, B. Maggs, S. Rao, A. Rosenberg, Work-preserving emulations of fixed-connection networks, in: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 227–240.
- [7] T. Leighton, B. Maggs, R. Sitaraman, On the fault tolerance of some popular bounded degree networks, in: Proceedings of the 33rd Annual Symposium on Foundation of Computer Science, 1992, pp. 542–552.
- [8] C.E. Leiserson, S. Rao, S. Toledo, Efficient out-of-core algorithms for linear relaxation using blocking covers, in: Proceedings of the 34th Annual Symposium on Foundation of Computer Science, 1993, pp. 704–713.
- [9] M.O. Rabin, Efficient dispersal of information for security, load balancing and faults tolerance, J. ACM 36 (2) (1989) 335–348.
- [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, Comm. ACM 33 (8) (1990) 103–111.
- [12] L.G. Valiant, General purpose parallel architectures, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990.