AN OPTIMAL SYNCHRONIZER FOR THE HYPERCUBE

(Extended Abstract)

David Peleg §‡ and *Jeffrey D. Ullman* §

Department of Computer Science
Stanford University, Stanford, California

Abstract

The synchronizer is a simulation methodology introduced by Awerbuch [A1] for simulating a synchronous network by an asynchronous one, thus enabling the execution of a synchronous algorithm on an asynchronous network. In this paper we present a novel technique for constructing network synchronizers. This technique is developed from some basic relationships between synchronizers and the structure of a *t-spanning subgraph* over the network. As a special result, we obtain a synchronizer for the hypercube with optimal time and communication complexities.

1. Introduction

Algorithms for synchronous networks are easier to design, debug and test than similar algorithms for asynchronous networks. The behavior of asynchronous systems is typically harder to grasp and analyze. Consequently, it is desirable to have a uniform methodology for transforming an algorithm for synchronous networks into an algorithm for asynchronous networks. This tool will enable one to design an algorithm for a synchronous network, test it and analyze it in that simpler environment, and then use the standard methodology to transform the algorithm into an asynchronous one, and use it in the asynchronous network.

This general approach for handling asynchronicity was introduced by Awerbuch in [A1], and referred to as a *synchronizer*. His paper proposes several specific methods for implementing a simulation of this type, and studies their complexities. Awerbuch later [A2] demonstrated another important and perhaps surprising use of the synchronizer, namely, the design of asynchronous algorithms for various problems, that were more efficient than any previously known. One might argue that in order to achieve a fast asynchronous algorithm it is necessary to program it directly in the environment in which it is to be run (much the same as programming in low-level languages usually yields better performance than compiling a high-level program). However, the inherent difficulty in reasoning about an asynchronous network sometimes makes it hard to reach an optimal solution for a problem directly in such an environment. There-

‡ Supported in part by a Weizmann fellowship

§ Supported in part by contract ONR N00014-85-C-0731

fore the approach of solving the problem in a synchronous environment and then transforming it into an asynchronous solution may yield a more efficient algorithm. This phenomenon was demonstrated on several graph problems, such as breadth-first-search and maximum flow [A2], for which algorithms obtained by combining a standard synchronous algorithm with a synchronizer yield an asynchronous algorithm with better performance (in terms of time and/or number of messages) than all previously known ones.

It is clear that every synchronizer incurs some time and communication costs for the synchronization of every round. Let us denote the time and communication requirements added by a synchronizer ν for each pulse (time step) of the synchronous algorithm by $T(\nu)$ and $C(\nu)$, respectively. Clearly, an efficient synchronizer should keep these costs as low as possible. In [A1], Awerbuch presents three synchronizers, named α, β and γ. These synchronizers demonstrate a certain trade-off between their communication and time requirements.

The first two simple synchronizers represent the two end-points of this trade-off. Synchronizer α is time optimal, i.e., $T(\alpha) = O(1)$, but its communication complexity is $C(\alpha) = O(|E|)$. On the other hand, synchronizer β achieves an optimal number of messages (i.e., $C(\beta) = O(|V|)$), but its time requirement may be high, namely, $T(\beta) = O(D)$. Here V, E and D denote the set of vertices, the set of edges and the diameter of the network, respectively. (Note that these complexities are crucial since they represent the overhead *per pulse* of the synchronous algorithm.) The third, more involved synchronizer γ is a combination of the two previous ones, which achieves some reasonable middle points on this scale. In particular, it is possible to achieve $C(\gamma) = O(k|V|)$ and $T(\gamma) = O(\log_k |V|)$, where k is a parameter taken from the range $2 \leq k < |V|$.

Awerbuch proves also some lower bounds which demonstrate that for some networks, the best possible improvements are within constant factors from synchronizer γ [A1, A3]. However, this lower bound is not global, and for various networks one can do better. For example note that for the class of bounded-degree networks, synchronizer α is optimal in both time and messages, since $|E| \in O(|V|)$. This covers many common architectures proposed for parallel computing, like meshes, butterflies and cube-connected cycles, rings etc. The same holds also for the class of trees and planar graphs in general. Likewise, for bounded-diameter networks, synchronizer β is optimal. Thus the problem remains interesting only for graph classes in-between.

A notable example of a network for which the current solutions are not satisfactory is the hypercube. The hypercube is a well-known network topology, for which a rich class of parallel algorithms can be implemented efficiently (cf. [P,U]). This class includes basic communication primitives like sorting and routing as well as computational algorithms such as discrete Fourier-transforms and similar problems. Some recent studies dealt with the fault-tolerance properties of the hypercube [BS, DHSS].

It is possible to construct synchronizers of type α, β or γ for the hypercube using the constructions of [A1], but the resulting complexities are not optimal. Furthermore, it is interesting to note that the hypercube has an even better synchronizer of type γ, which can be obtained by direct construction (rather than by using the algo-

rithm of [A1]) and whose complexities are $T = O(\log \log V)$ and $C = O(V)$, which is still not optimal. The main result of this paper is the construction of an optimal synchronizer [with $T = O(1)$ and $C = O(|V|)$] for the hypercube.

On our way towards this goal, we introduce a novel general methodology for a synchronizer (which we term δ, for consistency purposes). This methodology is developed by exploiting the close connection between synchronizers and the structure of a *t-spanner* on a network. Given a network $G = (V, E)$, a subgraph $G' = (V, E')$ is a *t-spanner* of G if for every $(u, v) \in E$, the distance between u and v in G' is at most t.

In the sequel we derive the following basic connections between t-spanners and synchronizers.

Theorem 1:

1. *If the network G has no t-spanner with at most m edges, then every synchronizer ν for G requires either $T(\nu) \geq t + 1$ or $C(\nu) \geq m + 1$.*
2. *If the network G has a t-spanner with m edges, then it has a synchronizer δ with $T(\delta) = O(t)$ and $C(\delta) = O(tm)$.* ∎

In fact, the lower bounds of [A1,A3] implicitly use part 1 of Theorem 1.

We then proceed to prove the existence of a 3-spanner with a linear ($O(2^n)$) number of edges for the hypercube of dimension n (henceforth, the n-cube). This yields the desired optimal synchronizer for the hypercube.

Theorem 2: *For every $n \geq 0$, the n-cube has a synchronizer of type δ with optimal time and communication complexities $T(\delta) = O(1)$ and $C(\delta) = O(2^n)$.* ∎

The rest of the abstract is organized as follows. Section 2 contains the definitions of the synchronous and asynchronous models, and Section 3 gives an overview of the basic structure of a synchronizer. Finally in Section 4 we discuss the relationships between synchronizers and t-spanners and prove Theorem 1, and in Section 5 we construct an optimal synchronizer of type δ for the hypercube and thus prove Theorem 2.

2. The Model

We consider the standard model of an asynchronous point-to-point communication network (e.g. [A1, GHS] etc.). The network is described by an undirected graph $G = (V, E)$. The nodes of the graph represent the processors of the network and the edges represent bidirectional communication channels between the processors.

All the processors have distinct identities. There is no common memory, and algorithms are event-driven (i.e., processors cannot access a global clock in order to decide on their action). Messages sent from a processor to its neighbor arrive within some finite but unpredictable time. Each message contains a fixed number of bits, and therefore carries only a bounded amount of information.

A synchronous network is a variation of the above model in which all link delays are bounded. More precisely, each processor keeps a local clock, whose pulses must satisfy the following property. A message sent from a processor v to its neighbor u at pulse p of v must arrive at u before pulse $p + 1$ is generated by u.

Our complexity measures are defined as

follows. The *communication complexity* of an algorithm A, C_A, is the worst-case total number of messages sent during the run of the algorithm. The *time complexity* of an algorithm A, T_A, is defined as follows. For a synchronous algorithm, T_A is the number of pulses generated during the run. For an asynchronous algorithm, T_A is the worst-case number of time units from the start of the run to its completion, assuming that each message incurs a delay of at most one time unit. This assumption is used only for performance evaluation, and does not imply that there is a bound on delay in asynchronous networks.

Next, let us formally define the hypercube of dimension n, $H_n = (V_n, E_n)$. This network is defined by $V_n = \{0,1\}^n$ and

$$\begin{aligned} E_n &= \{(x,y) \mid x,y \in V_n, \\ &\quad x \text{ and } y \text{ differ in exactly one bit}\}. \end{aligned}$$

The network has $|V_n| = 2^n$ nodes, $|E_n| = n \cdot 2^{n-1}$ edges, and diameter n.

Finally let us define the *Cartesian product* of two graphs. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The Cartesian product of G_1 and G_2, denoted $G_1 \times G_2$, is defined as

$$G_1 \times G_2 = (V_1 \times V_2, E)$$

where

$$\begin{aligned} E &= \{((u_1,v_1),(u_2,v_2)) \mid \\ &\quad (u_1 = u_2 \text{ and } (v_1,v_2) \in E_2) \text{ or} \\ &\quad (v_1 = v_2 \text{ and } (u_1,u_2) \in E_1)\}. \end{aligned}$$

Hence $G_1 \times G_2$ is constructed by substituting a copy of G_2 for each vertex in G_1 and drawing in edges between corresponding nodes of adjacent copies.

The H_n hypercube can be viewed as the Cartesian product graph $H_n = H_{n-k} \times H_k$, for any $0 \leq k \leq n$. Later, we make use of this characterization of the cube.

3. Synchronizers

The synchronizer is intended to enable any synchronous algorithm to run on any asynchronous network. The goal of the simulation is generating a sequence of local clock pulses at each processor of the network, satisfying the following property: pulse number p is generated by a processor only after it received all the messages of the algorithm sent to it by its neighbors during their pulse number $p-1$.

This property is easy to guarantee as long as we restrict our attention to synchronous algorithms with complete communication, i.e., algorithms that requires every processor to send messages to every neighbor at every time pulse. The obvious problem with partial communication algorithms is that in case processor v did not send any message to its neighbor u at a certain pulse, u is obliged to wait forever for a message, as link delays in the asynchronous network are unpredictable.

The conceptual solution proposed in [A1] consists of two additional phases of communication. The first phase simply requires every processor receiving a message from a neighbor to send back an acknowledgement. This way, every processor learns, within finite time, that all the messages it sent during a particular pulse have arrived. Such a processor is said to be *safe* with respect to that pulse. Note that introducing this phase does not increase the message complexity or the time complexity of the algorithm by more than a constant factor.

A processor may generate a new pulse whenever it learns that all its neighbors are safe with respect to the current pulse. Thus the second and main phase of the synchronizer involves delivering this "safety" information. This phase is thus responsible for

the additional time and message requirements, denoted earlier by $C(\nu)$ and $T(\nu)$, for a synchronizer ν.

The complexities of a synchronous algorithm S are related to those of the asynchronous algorithm A resulting from combining S with a synchronizer ν by $C_A = C_S + T_S \cdot C(\nu)$ and $T_A = T_S \cdot T(\nu)$. (We ignore, for the purposes of the present discussion, any additional costs of an initialization phase which may be needed for setting the synchronizer up.)

Let us demonstrate these ideas by giving a brief description of the three synchronizers introduced in [A1]. Synchronizer α is the simplest. After the execution of a certain pulse, when a processor learns that it is safe, it simply reports this fact directly to all its neighbors. Thus the behavior and complexity of this synchronizer boil down to those of an algorithm in which every processor sends messages to every neighbor in every pulse, so $C(\alpha) = O(|E|)$ and $T(\alpha) = O(1)$.

For synchronizer β we assume the existence of a rooted spanning tree in the network. After the execution of a certain pulse, the safety information is collected "bottom-up" on the tree; whenever a processor learns that it and all its descendants in the tree are safe, it reports this fact to its parent. Eventually the root learns that all the processors in the network are safe, and then it broadcasts this fact along the tree, letting the processors start a new pulse. Since the process is carried out on the tree, the complexities of synchronizer β are $C(\beta) = O(|V|)$ and $T(\beta) = O(H)$, where H is the height of the tree. In the worst case $H = O(|V|)$ too.

The last synchronizer, γ, is based on a combination of the previous ones, in which synchronizer β is applied to certain "clusters" of processors and synchronizer α is used to synchronize the clusters. We omit a precise description of this construction.

4. Synchronizers and t-spanners

In this section we prove the two parts of Theorem 1. This amounts to showing two complementary relationships between t-spanners and synchronizers. On the one hand, the nonexistence of a t-spanner of a certain size implies a lower bound on the complexities of any synchronizer for the network. On the other hand, the existence of a t-spanner can be used for constructing a synchronizer of a new type, δ.

Theorem 1 (Part 1): *If the network G has no t-spanner with at most m edges, then every synchronizer ν for G requires either $T(\nu) \geq t+1$ or $C(\nu) \geq m+1$.*

Proof: Let us assume that the network $G = (V, E)$ has no t-spanner of m edges, and yet it has a synchronizer ν with $C(\nu) \leq m$, i.e., using m or fewer messages per pulse. The argument is similar to that of the lower bounds of [A1, A2]. The requirement from the synchronizer is to ensure that a processor does not produce a new pulse before it gets all the messages sent to it in the previous pulse. Thus, between every two consecutive pulses there must be some transfer of information between each pair of neighbors in the network. Otherwise, the completely asynchronous nature of the network will force these neighbors to wait forever for a message that may still be on its way. Consider the set E' of edges through which the messages of the synchronizer were sent. The information flow between every pair of neighbors in G has to go through the edges of E'. Since only m or fewer messages were sent, the number of these edges is at most

m. Therefore by hypothesis, the subgraph $G' = (V, E')$ is not a t-spanner. This implies that there is an edge $(u, v) \in E$ such that the distance between u and v in G' is at least $t+1$. Thus, the information flow between u and v may require $t+1$ time units, so the time complexity of the synchronizer ν is at least $t+1$. ∎

Theorem 1 (Part 2): *If the network G has a t-spanner with m edges, then it has a synchronizer δ with $T(\delta) = O(t)$ and $C(\delta) = O(tm)$.*

Proof: Assume that the network $G = (V, E)$ has a t-spanner $G' = (V, E')$ of m edges. The synchronizer δ for G is constructed as follows. The safety information is transmitted over the spanner in rounds, which may be viewed as "subpulses". When a processor v learns that it is safe, it sets a counter c to 0 and sends the message "safe" to all its neighbors *in the spanner*. Then it waits to hear a similar message from all these neighbors. Upon receiving a "safe" message from all neighbors (again - in the spanner), it increases its counter and repeats the process (i.e., it sends the message "safe" again, and then waits for similar messages and so on). This is done for t rounds. When $c = t$, the processor v may generate its next pulse.

Lemma 4.1: *When v holds $c = i$, every processor u at distance i or less from v in G' is safe.*

Proof: By induction on i. For $i = 0$ the claim is immediate, as v reaches this stage of the synchronizer only after it is safe itself. Now consider the time when v increases c to $i+1$. This is done after v received $i+1$ "safe" messages from every neighbor in G'. These neighbors each sent the $(i+1)$st message only after having $c = i$. Thus, by the inductive hypothesis, for every such neighbor u, every processor w at distance i or less from u in G' is safe. Thus every processor w at distance $i+1$ or less from v in G' is safe too. ∎

Corollary 4.2: *When v holds $c = t$, every neighbor of v in G is safe.*

Proof: By the lemma, when v holds $c = t$, every processor u at distance i from v in G' is safe. By the definition of t-spanners, every neighbor of v in G is at distance i or less from v in G'. Thus every such neighbor is safe. ∎

It is clear that the time delay of synchronizer δ is at most $O(t)$, and the number of additional messages is $O(mt)$. Thus the proof is complete. ∎

The synchronizers of [A1] do not use spanners explicitly, and their formulations are quite different from our synchronizer δ. Nevertheless it is interesting to note that their underlying structure is strongly related to the notion of spanners. For instance, synchronizer α is based essencially on the fact that every graph is its own 1-spanner, and the construction algorithm of [A1] for synchronizer γ establishes the fact that every graph has an $O(\log n)$-spanner with linear number of edges.

5. An Optimal Synchronizer δ for the Hypercube

In this section we show how to construct a 3-spanner for the hypercube H_n with a linear $[O(2^n)]$ number of edges. This in turn implies an optimal synchronizer δ and thus proves our Theorem 2.

A *dominating set* for a graph G is a

subset U of vertices with the property that for every vertex v of G, U contains either v itself or some neighbor of v.

To construct dominating sets, we make use of the notion of a Hamming code, a well-known idea from coding theory (cf. [H]). Since the Hamming code is so central to what we do, we review its definition and properties. The reader familiar with Hamming codes can skip directly to Lemma 5.1.

A string of 0's and 1's is a *word*. The *product* of two code words of the same length is their bitwise logical "and"; their *sum* is their bitwise exclusive or. For example, the product of 0011 and 0101 is 0001, and their sum is 0110. The *weight* of a code word is the number of 1's in the word.

For any integer r, the Hamming code $HC(r)$ is defined by making reference to a matrix of 0's and 1's with r rows and $2^r - 1$ columns. The columns are all the binary integers from 1 up to $2^r - 1$. For example, here is the matrix used to define $HC(3)$:

$$M_3 = \begin{matrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix}$$

The *code words* in $HC(r)$ are all those words of length $2^r - 1$ whose product with each of the rows of the defining matrix M_r has even weight. For example, 0000000 and 0101010 are code words of $HC(3)$. Word 0000000 has a product with each row of M_r that has weight 0, while 0101010 has products of weight 2, 2, and 0, respectively.

The *distance* between two words is the weight of their sum. Put another way, the distance is the number of positions in which the two words differ. For example, each pair of rows of M_r has distance 4.

An important and easily proved fact about Hamming codes is that the sum of two code words is also a code word. As a consequence, the Hamming codes have minimum distance 3; that is, no two code words have distance less than 3. In proof, let u and v be code words, and let w be their sum, which is also a code word. If the weight of w were 1, then since the product of w with every row of M_r has even weight, those products would have to have weight 0; i.e., the one column in which w has a 1 would have all 0's in M_r. Since there is no such column, w cannot have weight 1. Similarly, if w had weight 2, we could argue that two columns of M_r would have to be identical, which is not the case.

Now, let the *neighborhood* of a code word be that word plus all words of distance 1. Then in $HC(r)$, all neighborhoods have 2^r members. No word can be in the neighborhood of two code words, or else those code words would be of distance 2.

The last fact we need about Hamming codes is that $HC(r)$ has exactly 2^{2^r-1-r} members. The argument is that the rows of M_r can be thought of as independent vectors in the vector space of $2^r - 1$ dimensions over the field of two elements. $HC(r)$ is thus the null space of the vector space (of dimension r) whose basis is the rows of M_r, and therefore $HC(r)$ is of dimension $2^r - 1 - r$.

The number of words in the neighborhood of some code word of $HC(r)$ is thus $2^r \times 2^{2^r-1-r}$, or 2^{2^r-1}. This is exactly the number of binary words of length $2^r - 1$. Thus, every binary word of length $2^r - 1$ is in the neighborhood of exactly one code word of $HC(r)$. Put another way, $HC(r)$ is a dominating set for the hypercube H_{2^r-1}.

Lemma 5.1: *For every $n \geq 1$, the n-cube has a dominating set of at most $\frac{2^{n+1}}{n}$ nodes.*

Proof: Let us first consider the case of $n = 2^r - 1$ for some $r \geq 1$. Let U be the Hamming code $HC(r)$. As we argued above, U is a dominating set for the n-cube, and $|U| = \frac{2^n}{n+1} \leq \frac{2^{n+1}}{n}$.

Now consider an arbitrary $n \geq 1$. Let r be the integer satisfying $2^r - 1 \leq n < 2^{r+1} - 1$ and let $d = 2^r - 1$. Note that $n/2 \leq d$. Let U_d be a dominating set for the d-cube. View the n-cube as the Cartesian product $H_n = H_{n-d} \times H_d$, and let $U = \{(x, y) \mid x \in V_{n-d},\ y \in U_d\}$, where V_{n-d} is the set of vertices of H_{n-d}. Clearly U is a dominating set for the n-cube, and its size is $2^{n-d}|U_d| = 2^{n-d}\frac{2^d}{d+1} = \frac{2^n}{d+1} \leq \frac{2^{n+1}}{n}$. ∎

We finally construct a spanner for the n-cube. The cases $n = 1, 2$ can easily be verified directly, so assume $n \geq 3$. Our construction uses dominating sets for some of the subcubes of our hypercube. Consider an n-cube H_n, and let $p = \lfloor \frac{n}{2} \rfloor$ and $q = \lceil \frac{n}{2} \rceil$. View H_n as the Cartesian product $H_p \times H_q$. Let U_1 and U_2 be minimum-size dominating sets for H_p and H_q, respectively.

We define the subgraph $G' = (V_n, E')$ by choosing the following sets of edges:

1. Every edge $((x, y), (x, y'))$ s.t. $(y, y') \in E_q$ and $y' \in U_2$.
2. Every edge $((x, y), (x', y))$ s.t. $(x, x') \in E_p$ and $x' \in U_1$.
3. Every edge $((x, y), (x, y'))$ s.t. $(y, y') \in E_q$ and $x \in U_1$.
4. Every edge $((x, y), (x', y))$ s.t. $(x, x') \in E_p$ and $y \in U_2$.

Lemma 5.2: *G' has fewer than $7 \cdot 2^n$ edges.*

Proof: By the previous lemma $|U_1| \leq \frac{2^{p+1}}{p}$ and $|U_2| \leq \frac{2^{q+1}}{q}$. Therefore the number of edges $(y, y') \in E_q$ with $y' \in U_2$ is at most $q|U_2| \leq 2^{q+1}$, and so the number of edges of the first type is at most $2^p 2^{q+1} = 2^{n+1}$. Similarly the number of edges of the second type is at most 2^{n+1}. For the third type we get the bound $|U_1| \cdot |E_q| \leq \frac{2^{p+1}}{p} \cdot q \cdot 2^{q-1} \leq \frac{q}{p} \cdot 2^n$, and similarly of the fourth type there are at most $|U_2| \cdot |E_p| \leq \frac{p}{q} \cdot 2^n$ edges. (This simple counting ignores multiple occurrences of certain edges in the various types, which in fact implies a slightly smaller bound.) Since $n \geq 3$, $\frac{p}{q} + \frac{q}{p} < 3$, so overall we get a bound of fewer than $7 \cdot 2^n$ edges ($6 \cdot 2^n$ for even n). ∎

Lemma 5.3: *G' is a 3-spanner of the n-cube.*

Proof: Consider two neighboring vertices u, v in H_n. Let $u = (x, y)$ where $x \in \{0, 1\}^p$. Then either $v = (x, y')$ for some neighbor y' of y in H_q or $v = (x', y)$ for some neighbor x' of x in H_p. In the first case, G' contains the following edges:

(1) $e_1 = ((x, y), (x', y))$ for some $x' \in U_1$ (type 2).

(2) $e_2 = ((x', y), (x', y'))$ (type 3).

(3) $e_3 = ((x', y'), (x, y'))$ (type 2).

These three edges constitute a path of length 3 in G' connecting u and v. The second case is handled similarly, using edges of types 1 and 4. ∎

Corollary 5.4: *For every $n \geq 0$ the n-cube has a 3-spanner of fewer than $7 \cdot 2^n$ edges.* ∎

Finally, using Theorem 1 together with the last corollary, we have

Theorem 2: *For every $n \geq 0$, the n-cube has a synchronizer of type δ with optimal time and communication complexities $T(\delta) = O(1)$ and $C(\delta) = O(2^n)$.* ∎

Acknowledgements

We wish to thank Alex Schaffer for helpful discussions.

References

[A1] B. Awerbuch, Complexity of Network Synchronization, *J. of the ACM* **32**, (1985), pp. 804–823.

[A2] B. Awerbuch, Reducing Complexities of the Distributed Max-Flow and Breadth-First-Search Algorithms by Means of Network Synchronization, *Networks* **15**, (1985), pp. 425–437.

[A3] B. Awerbuch, Communication - Time Trade-Offs in Network Synchronization, *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 272–276.

[BS] B. Becker and H-U. Simon, How Robust is the n-Cube?, *Proc. 27th Symp. on Foundations of Comp. Science*, 1986, pp. 283–291.

[DHSS] D. Dolev, J. Halpern, B. Simons and R. Strong, A New Look at Fault Tolerant Network Routing, *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 526–535.

[GHS] R.G. Gallager, P.A. Humblet and P.M. Spira, A Distributed Algorithm for Minimum Weight Spanning Trees, *ACM Trans. on Program. Lang. & Systems* **5**, (1983), pp. 66–77.

[H] R. Hill, *A First Course in Coding Theory*, Oxford Applied Mathematics and Computing Science Press, Oxford, 1986.

[P] N.C. Pease, The Indirect Binary n-Cube Microprocessor Array, *IEEE Trans. Comp.* **6**, (1977), pp. 458–473.

[U] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.