Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11th February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MATHEMATICAL CENTRE TRACTS 158

FOUNDATIONS OF COMPUTER SCIENCE IV

DISTRIBUTED SYSTEMS:

PART 1, ALGORITHMS AND COMPLEXITY

J.W. DE BAKKER (ed.)

J. VAN LEEUWEN (ed.)

MATHEMATISCH CENTRUM AMSTERDAM 1983

1980 Mathematics subject classification: 68C05, 68C25

1982 CR. Categories: B.7.1, B.7.2, D.1.3, D.3.3
ISBN 90 6196 254 4

Copyright © 1983, Mathematisch Centrum, Amsterdam

CONTENTS

Contents i

Preface ii

Authors' current addresses iii

J. VAN LEEUWEN: *Distributed computing* 1

L.G. VALIANT: *Parallel computation* 35

G.M. BAUDET: *Design and complexity of VLSI algorithms* 49

M.R. KRAMER & J. VAN LEEUWEN: *Systolic computation and VLSI* 75

D.P. DOBKIN: *VLSI, algorithms and graphics* 105

PREFACE

The 4th Advanced Course on the Foundations of Computer Science was held June 14-25, 1982, in Amsterdam as part of an international program of Advanced Courses sponsored by the CREST Subcommittee on Training in Data Processing of the Commission of the European Communities.

The Advanced Courses on the Foundations of Computer Science are organized to provide an opportunity for computer science graduates and professionals to learn about the modern developments in theoretical computer science at a high level. The 4th Advanced Course was devoted in particular to Distributed Systems and Computation. Six distinguished lecturers were invited to present a series of six lectures on leading issues and new results in their current field of specialty. Furthermore, a number of lectures by the directors were included in the program of the course.

These volumes contain the (edited) text of the lectures given on the occasion of the 4th Advanced Course. The material, mostly written especially for the Course, usually presents an original view of an entire research area which is not available in this form yet from textbooks for classroom use. We believe that the chapters will serve as a valuable source of material for high-level seminars in theoretical computer science.

We thank the lecturers for their excellent contributions and the participants for being a most receptive audience. We are very grateful to the Dutch Ministry of Education and the Commission of the European Communities, which together provided the necessary funds for organizing the Course. Finally, we thank Mrs. S.J. Kuipers-Hoekstra for her invaluable assistance throughout the organization of the Course and the Publication Service of the Mathematical Centre for the technical realization of these volumes.

J.W. de Bakker - J. van Leeuwen
Directors of the Course

AUTHORS' CURRENT ADDRESSES

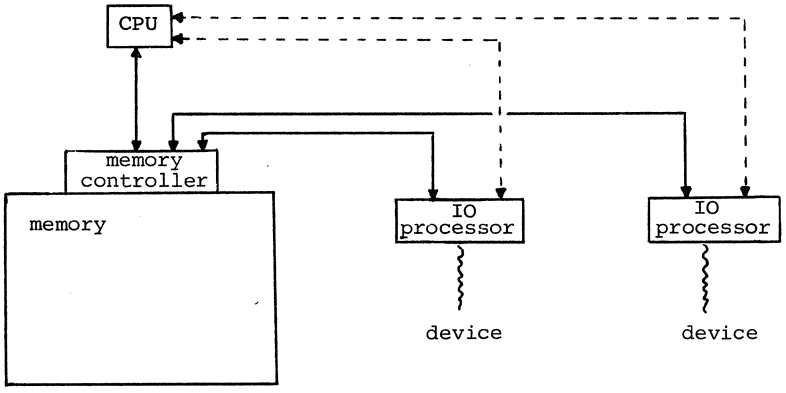
G.M. Baudet	Department of Computer Science, Brown University, Providence, RI 02912, USA
D.P. Dobkin	Electrical Engineering and Computer Science Department, Princeton University, Princeton, NJ 08544, USA
M.R. Kramer	Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, the Netherland
J. van Leeuwen	Vakgroep Informatica, Rijksuniversiteit Utrecht, Princeton-plein 5, Postbus 80.002, 3508 TA Utrecht, the Netherlands
L.G. Valiant	Aiken Computation Laboratory, Harvard University, Cambridge,MA 02138, USA

DISTRIBUTED COMPUTING

J. van Leeuwen
University of Utrecht, Utrecht, the Netherlands

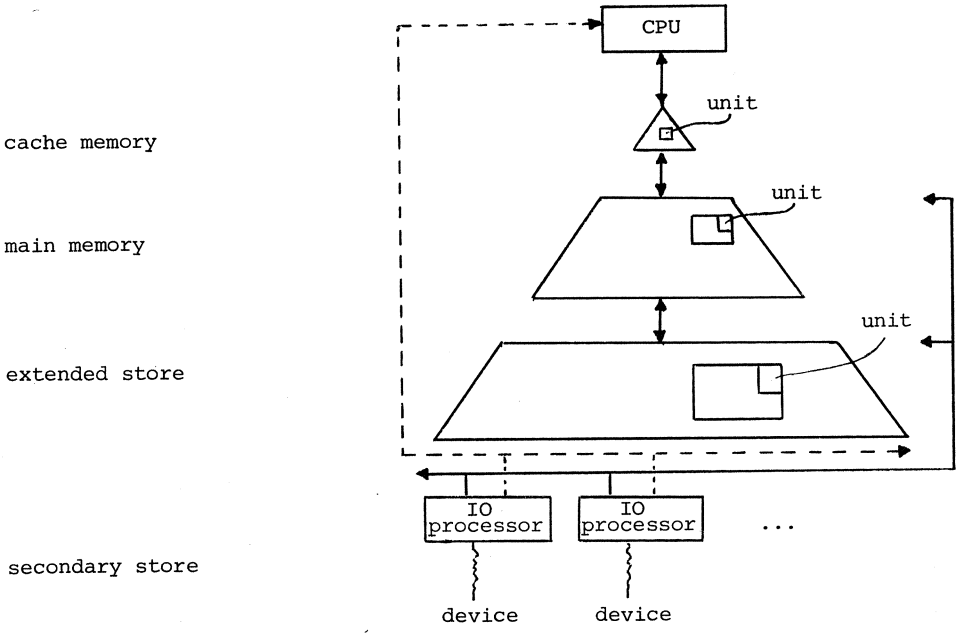
1. Introduction. Distributed architectures are steadily advancing and will eventually replace conventional computer designs built around a single central processor. In these notes I shall attempt to describe the trends in the theoretical investigation of the problems that arise in distributed information processing. The subject is by no means new. Most computer systems today can be regarded as distributed systems in certain respects. Only in recent years the impetus from VLSI-technology (in the small) and local and wide area networks (in the large) has added further to the significance of distributed processing, as a viable alternative to the physical limitations of even the largest single processor systems and the inordinate investments that they require. With hardware costs declining and commercially supported interconnection technologies now available it might be more economical indeed to achieve high performance by utilizing dedicated computing units working independently in parallel, rather than through the use of extremely complex high speed single components. In many ways though, distribution seems to create more problems than it solves. As we go along I shall try to point out some of the insights in distributed computing that have emerged in the past decade or so, in a brief but comprehensive survey of the area. For references see also [1].

2. Early developments in hardware. The earliest computers consisted of a central memory and a single CPU to execute a stored program of instructions from a limited repertoire. No concurrency of operation was provided for between computations by the CPU and I/O, resulting in an unnecessary idling of the CPU while a data transfer was taking place on much slower hardware. Subsequent introduction of separate I/O processors (channels, peripheral processing units) into all architectures enabled the CPU to delegate I/O commands and switch to another computation in the meantime. I/O processors



would report by sending an interrupt. The separation of computation and I/O processing made it possible to load and execute a number of jobs simultaneously with obvious advantages of mixing compute-bound and I/O-bounds jobs in a suitable manner. The need to schedule and service interrupts for optimal performance is one of the earliest problems in distributed computing. Buffering (and the ultimate form of it, virtual devices) led to typical producer-consumer problems.

Likewise, early computers exhibited no concurrency of operation between computations by the CPU (i.e., instruction execution) and instruction fetching. Later CPUs were pipelined, which allowed for the initiation of a next instruction while others were still progressing through the circuitry. To facilitate a rapid transfer of information to and from the CPU, memory got decomposed into fast parts (small and expensive) and slower parts (larger and cheaper). A typical memory hierarchy consists of the following parts:



Program code was split and distributed over the hierarchy (blocks, segments, pages, caches) with the most immediately needed instructions highest in the "pyramid". Assumptions of locality led to a great variety of fetch/replacement strategies for units in the hierarchy, all aimed at minimizing the chance of unit faults throughout the hierarchy. Manual and automatic techniques for program restructuring (to improve locality) were proposed to allow for smooth transmissions up and down the hierarchy.

3. Multiprogramming. The variety of programs processed by a general purpose computer system is such that (with a suitable admission policy or job scheduling) every job requires only a small portion of the available resources and different jobs can do with different portions. A system configured to meet or exceed the joint resource demands of many jobs simultaneously is likely to yield a greater throughput (and thus, a greater cost-effectiveness) as long as the overhead in managing the resources by the operating system does not annihilate the expected gain. Multiprogramming is a technique (or rather, a set of techniques) to execute a number of programs "simultaneously"

provided their total resource requirements at any time are not greater than the total available on the system. Multiprogramming in particular allows for the interleaved or concurrent execution of a set of (usually independent) user programs and (highly dependent) standard routines for system management on a single CPU.

Multiprogramming implies the need to share resources (CPU, memory, devices) and thus to distribute their availability to programs over time. To coordinate the sharing, mechanisms are required by which the programs can communicate their needs. The system may have to communicate back to the programs, as in present day interactive and real-time applications.

4. Concurrency. User jobs and system routines generally are independent units (tasks) that could proceed in parallel if sufficiently many processors were available. In industrial applications, in fact, several jobs may have to be performed at the same time, with critical developments in one job perhaps affecting (interrupting) the work in others. Any implementation of concurrency requires mechanisms that permit the sharing of common resources and mechanisms that enable concurrently executing units to exchange information (communication, cooperation) or to coordinate their action (synchronization). No implementation can be proved sound unless there is an adequate underlying model of concurrent processing.

5. Processes. Each independent unit of execution (task) in a system may be called a process. In the sixties E.W. Dijkstra and others promoted the view that concurrent activity can best be modelled by a set of cooperating processes which alternate between independent activity and periods of communication. The process thus became the "unit" of concurrency. Much research has focused on the issues of interprocess communication and synchronization.

Processes may be activated by (i) calling a static copy, (ii) resuming it (as a coroutine), (iii) "forking" in a sequential program, or (iv) dynamic creation, with an implicit hierarchical (parent-child) ordering. Processes may be de-activated by (i) termination, (ii) blocking or voluntary transfer of control, (iii) "joining", or (iv) destruction and transfer of control.

Processes communicate by signals (wait/post, lock/unlock), shared variables (e.g. semaphores) or messages (mailboxes, interprocess queues). To obtain exclusive access to shared resources more transparent and structured primitives have been provided (critical regions, monitors).

Processes requiring exclusive access to a shared variable or resource must compete for it. Any mechanisms used for it must guarantee mutual exclusion and (normally) bounded waiting. In addition resources should be given out wisely to prevent deadlock. Instead of competing among each other, processes could simply submit their requests to a manager or monitor that is put in charge of (i.e., encapsulates) the shared resource.

The process notion has had a tremendous impact on system structuring. Given an implementation of suitable primitives for process creation/destruction and interprocess communication (in a nucleus or kernel of the system) higher level processes can be conceived that use (share!) the facilities provided by lower level processes and managers, which eventually lead to executable code on the available hardware processor. Every level thus provides a virtual multiprocessor for the processes at the next level. The approach has become dogmatic for all modular system design.

6. Semantics of processes. Any system of concurrently computing units is obtained by connecting processes (explicitly of implicitly), and insisting on a protocol for interprocess communication. Petri nets (see e.g. [2]) are among the earliest frameworks that were introduced to model the distributed flow of control among multiple processes occurring concurrently. A Petri net consists of places (which can hold tokens) and transitions (requiring tokens for control transfers), connected by arcs. A transition "fires" by taking one token from every place at the origin of its incoming arcs (provided every one holds at least one token) and sending one token to every place at the end of its outgoing arcs. Petri nets feature many properties of concurrent computation including (i) nondeterminism (if at any time more than one transition is enabled, then any choice of them may fire), (ii) conflict (transitions may connect to common places and firing some may disable others) and (iii) asynchronism (there is no notion of time and thus no unique ordering of events). Deadlock-freeness (also called liveness) can be formulated as the requirement that in all reachable markings every transition is potentially firable. Hack

[3] proved that the liveness problem is recursively equivalent to the reachability problem and (thus) the problem is decidable, given the recent solution of the latter by Mayr [4] (see also Kosaraju [5]).

To adequately model the flow of data in a concurrent computation many different frameworks have been proposed. In a model due to Kahn [6] processes are viewed as sequential programs (procedures) with in-ports and out-ports that communicate data over fixed lines. It is assumed that this is the only way in which processes communicate, and that data sent always arrives in a finite but unpredictable time. The communication lines can be thought of as pipes or queues between processes. The possibly infinite sequence of data items passing any observer on a given line is called the history of the line. Kahn's theory is based on the view that processes are functions from histories (of the input lines) to histories (of the output lines) and that the behaviour of the net is described once all histories are known. Let sequences be defined over a domain D and partially ordered by the prefix relation. Processes $f : D^{\omega}x \ldots \rightarrow D^{\omega}x \ldots$ must be (i) monotone (i.e., more input only leads to additional output, or $u \leqq v \Rightarrow f(u) \leqq f(v)$) and (ii) continuous (no output after an infinite amount of input is received, or $f(\lim u_i) = \lim f(u_i)$). Histories can now be defined by a system of equations of the form $(Y,\ldots) = f(X,\ldots)$ where X,... are the histories of f's incoming lines and Y,... the histories of f's outgoing lines, and f ranges over all processes in the net. Given (monotone and) continuous f over cpo's like $D^{\omega}x \ldots$ Kleene's theorem asserts that such systems have a unique minimal solution, obtained by iterating the f's from $(\lambda,\lambda,\ldots)$. As a result, properties of nets that can be phrased in terms of histories may be proved by induction over their construction.

Intuitively Kahn's model fails to capture the nondeterminism inherent to concurrent computation, resulting e.g. when exclusive communication lines are absent. At some abstract level one could say that processes merely "act" (by changing state and sending messages) and that "events" take place at their ports (receipts of messages). Messages sent (as the result of an event) must eventually be received at their destination, i.e., turn up as some later event. In Hewitt's actor model (see e.g. [7]) parallel computation is thus modeled as a partial ordering of events, where events are said to be concurrent if there is no ordering relation between them. The following "laws" are believed essential to meaningful actor computation: (i) existence of a least element

(initial event), (ii) discreteness (the number of events between any two events is finite) and (iii) finite immediate successors (any single event can have only finitely many immediate successors).

In recent years many other attempts have been made to describe the composition of processes into nets and to reason about their composite behaviour with formalisms from logic (e.g. [8] and [9]), semantics (e.g. [10]) or language theory (e.g. [11]). Very recently Pratt [12] described composition as a closed operator on processes and (thus) elegantly solved the problem of defining the meaning of a composition of processes by specifying exactly what process is obtained by connecting processes together. Composition thus allows an algebraic study. All formalisms attempt at providing a sound (consistent) and/or complete proof system for reasoning about the corporate behaviour of processes such as fairness, deadlockfreeness and termination.

7. Languages for concurrent programming. Control of concurrent activity appears to be more difficult to achieve than control of sequential activity. Humans find it very hard, in general, to comprehend the combined effect of a number of activities which evolve simultaneously with independent speeds. For years programmers thought sequential,as suitable concepts and tools for parallel programming were lacking. Early primitives like the cobegin ... coend ([13]), the and ([14]) and the fork/join/quit statements have attempted to remedy this. More recently, the use of sets of guarded commands ([15]) was proposed as perhaps the most natural means for expressing concurrency. As processes were recognized as the unit of concurrency various communication primitives were proposed, initially to operate on shared variables. More structured notions like critical regions ([13], [16]) and conditional critical regions ([17]) and monitors ([18]) were introduced to replace low level synchronization commands of the P/V or receive/send variety. Pilot languages like MODULA ([19]), Concurrent Pascal ([20]) and Pascal Plus ([21]) incorporated suitable constructs for programming processes, monitors and queues of waiting processes. The requirements of mutual exclusion and fair scheduling were major problems to solve in an efficient manner.

More recently there is a trend in concurrent programming languages away from communication and synchronization through shared variables, and towards direct

communication between modules or processes. The former approach requires a common store and the latter does not, which thus seems more supporting for a view of processes as net-connected individual processors. Language designs like DP (Distributed Processes, [22]) and CSP (Communicating Sequential Processes, [23]) provided a testbed for several programming constructs. A CSP-program consists of a fixed and named set of disjoint parallel processes. Communication occurs by exchanging data in matching input/output commands. To this end processes P may contain statements of the form Q?<variable> (requesting an input from Q, to be assigned to <variable>) or Q!<expression> (send the value of <expression> to Q). Execution proceeds only after a valid rendez-vous has taken place. The use of guarded commands adds a tremendous flexibility to processes, but leads to every imaginable problem of nondeterminism and non-functionality.

MODULA-2 ([24]) has adopted similar views of communication by exchanging (importing and exporting) data, and simply lists in the specification of a module or process which identifiers must be "passed" to aid the linking of processes. The process concept is kept extremely simple (essentially that of coroutines) and is built on a system kernel that provides the type PROCESS and primitives NEWPROCESS (turns a procedure and its workspace into a named process), TRANSFER (suspends the current process and transfers control to another) and IOTRANSFER (like the former, but with an implicit transfer of control back to the suspended process upon an IO-complete interrupt). The programming language ADA (see e.g. [25]), again, is not tied to a single processor environment and offers a sophisticated facility for describing parallel activities (tasks) very much in the spirit of CSP.

8. Semantics of concurrent programs. Proving properties of concurrent programs is generally considered hard. Correctness, termination and other properties of interest for a system of parallel processes that are cooperating towards some goal do not immediately follow by straight application of the techniques of e.g. Hoare [26] known for sequential programs. New issues to cope with are partial ordering and communication between distinct units. Proof methods may be obtained when a suitable, algebraic notion of composition of processes is used (see 6). An important step forward in understanding parallelism has

resulted from the work of Owicki [27]. She proposed to proceed in two steps: (i) prove the properties of each process as a sequential program disregarding completely parallel execution and (ii) show that the execution of one process does not interfere with (i.e., does not destroy) the proof of the properties of another. The rationale is that if parallel execution does not invalidate the proofs, it cannot destroy the desired properties. An interesting application is given in [28]. More general, cooperating processes require some form of cooperating proofs. This has been expounded in the axiomatic proof theory now developed for CSP [29] and continues to be tested in other systems (see e.g. [30]).

9. Distributed systems. Breaking up programs or tasks into processes is a start toward multiple processor systems. Each process is a natural unit to allocate to an available processor. It is generally agreed that a distributed system exhibits the following characteristics (cf. [31]):

(i) it includes an arbitrary number of system and/or user processes,

(ii) the architecture is modular and consists of a possibly varying number of processing elements (PE's),

(iii) communication is achieved via some form of message passing over a shared communication structure (including perhaps shared memory),

(iv) some system-wide control is performed so as to provide for dynamic interprocess cooperation and runtime management,

(v) interprocess message transit delays are variable and some non-zero time always exists between the production of an event by a process (viz. a processing element) and the materialization of it at some intended destination.

Among the criteria used to compare distributed systems are their size or scale (viz. the distances over which messages are sent), the rates of data transfer and their degree of coupling. Coupling is said to be (i) strong if data transfer between the PE's is about as fast (say, ≧10 Mbps) as access of a PE to its own data, (ii) loose if PE's communicate through a channel comparable in speed to the transfer rate of secondary storage devices (say, .1-10 Mbps) and (iii) weak if PE's communicate through a channel of only a few Kbs (like a long distance telephone line). The distinction roughly

corresponds to (i) multi-processor systems, (ii) local area networks and (iii) wide area networks. We shall later see that there is a variety of interconnection structures (topologies) possible in each case.

10. VLSI systems (chips). Switching circuits are a natural model of distributed computing in the small, featuring many forms of parallelism and pipelining. With the advent of VLSI technology (see e.g. [32]) it has become possible to embed ("integrate") circuits of tens of thousands of components in the surface of a single chip. Special IO-pads (ports) along the boundary of the rectangular chip allow for data/signal transports to and from the environment, usually over a limited number of (multiplexed) pins into the hardwiring of some PC board. Rigorously simplifying the practical aspects of wiring and timing, Thompson [33] formulated a grid model of the chip surface to study the actual complexity of VLSI-circuits. Each cell may contain a single PE (another simplification!) or up to two orthogonal, crossing wires. The model has facilitated the study of a novel measure for circuits, its area A.

Definition. Given a connected graph $G = \langle V,E \rangle$ the minimum bisection width w of a set $S \subseteq V$ is the smallest number of edges that must be cut to split S into two isolated, equal halves.

The following result of Leighton [34] improves on an earlier theorem of Thompson. For the notion of crossing number, see [35].

Theorem. Let an n-node graph G have crossing number c and contain a set with minimum bisection width w. Then $A \geq c+n \geq \alpha.w^2$ for some fixed constant α.

Proof

Any embedding of G must contain n nodes and $\geq c$ crossings, thus $A \geq c+n$. Consider any drawing of G with c crossings. Turn every crossing into a "point" to obtain a planar graph G' with c+n points. The planar separator theorem ([36]) implies the existence of a constant β such that the original set can be bisected by dropping $\leq \beta.\sqrt{c+n}$ edges, thus by cutting at most this many edges in the original graph. Thus $w \leq \beta.\sqrt{c+n}$ and $A \geq c+n \geq \alpha w^2$, for $\alpha = 1/\beta^2$. □

Definition. Given an embedded circuit (a graph) $G = \langle V,E \rangle$ the minimum information flow I for a set $S \subseteq V$ is the minimum number of bits that must be exchanged between two halves of S, the minimum taken over all possible bisections of S.

Let the time T of a computation on a chip be measured in some reasonable way (e.g. the time between input of the first bit and output of the last).

Theorem. For any VLSI-circuit $AT^2 \geq \alpha I^2$, with α as before.

Proof

Let $S \subseteq V$ have minimum bisection width w, thus $A \geq \alpha.w^2$. The computation requires the transfer of $\geq I$ bits of information over the cut of w edges which takes $\geq I/w$ time. Hence $AT^2 \geq \alpha w^2 . (I/w)^2 = \alpha I^2$. □

Note that the results given are quite independent of the actual form of the chip. As I can often be estimated in a circuit-independent manner, the latter result suggests an "area-time" trade-off for VLSI-design. It can be shown e.g. that for DFT's of n b-bit integers $AT^2 = \Omega(b^2n^2)$. Useful techniques to prove it follow from [37]. Brent and Kung [38] have presented a detailed study of binary addition and multiplication in VLSI. For recent results, see e.g. [39]. Kramer and van Leeuwen [40, 41] have shown that wire routing and even deciding the embeddability of routable circuits in a given amount of area are NP-complete problems.

With the advent of techniques to produce multi-layered chips, it is of interest to explore the possible gains with a extra dimension to the layout problem. Thompson [33] (see also [42]) proved that a v-layered chip of area A can be embedded in the plane in $O(Av^2)$ area, thus making "planar" techniques of analysis applicable.

Chazelle and Monier [43, 44] have argued that under a more realistic assumption about communication times on a chip (linearity in distance traversed) much of the general theory for Thompson's model evaporates. In particular, asymptotically time and area notions are polynomially related to ordinary Turing machine time and space.

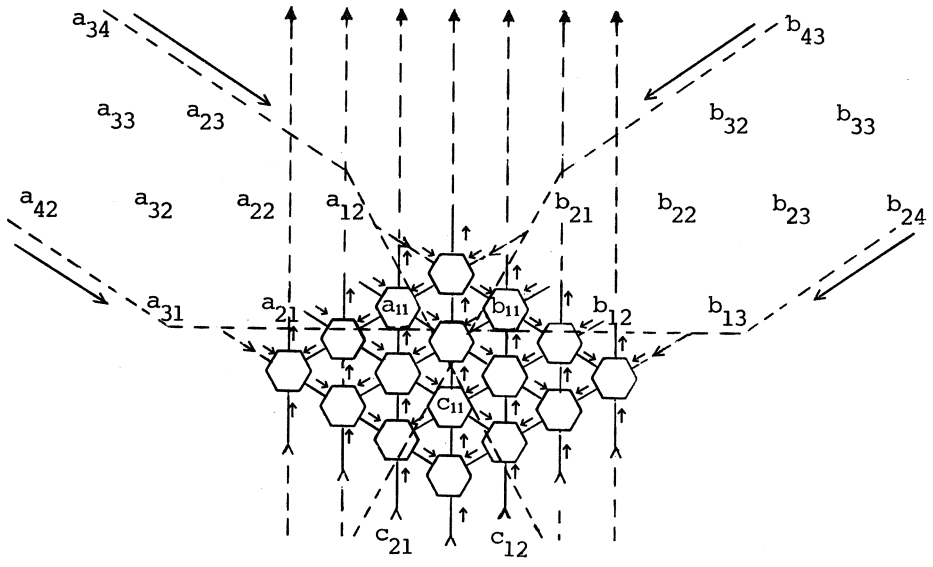
11. Systolic algorithms. Given the current technology it has become feasible to design chips for every imaginable special-purpose function in a system, an approach advocated by Kung [45] (see also [46]). The chip design must begin with a distributed algorithm design, conceptually specifying the overal structure of the PE's on the chip. The algorithm is a level of abstraction at which two aspects of the design can be contemplated: (i) the pattern of information flow between the PE's (including the number of cells needed, their placement and the movement of data between them) and (ii) the types of PE's and their timing. There are many similarities between modular programming and modular chip design: the design task must be broken into manageable subtasks with a well defined flow of information between them. Foster and Kung [46] identify the following properties for a "good" VLSI-algorithm:

(i) it can be implemented by means of only a few types of simple cells,

(ii) the data and control flow of the algorithm is simple and regular, allowing cells to be connected by a network with local and regular interconnections (like grids or hexagonal arrays),

(iii) the algorithm extensively employs pipelining and parallel processing.

Typically, the designs have several data streams move at constant velocity over fixed paths in the network, interacting at cells where they meet. In



this way many cells are kept active simultaneously and the computation hardly slows the data rate. Algorithms of this sort have been called systolic. Kung and Leiserson [47] and Kung [48] (see also [32]) offer many examples, usually for numeric computations, showing the versatility of the approach. See also [49].

Communication costs are a crucial factor that make systolic algorithms attractive. At every tick of some periodic clock communication can occur between a PE and its neighbors according to the communication graph only. Signals do not ripple on and are not broadcasted beyond the neighbors. In general, let the edges of a communication graph carry integer weights ≥ 0 indicating the time delay of signals along the corresponding line. To avoid race conditions we require of a "synchronous" system that every cycle in its communication graph has weight >0. Let G-1 be the graph obtained by reducing the weight of every edge in G by 1. Leiserson and Saxe [50] recently proved the following "systolic conversion theorem": if the G-1 of the communication graph G of a synchronous system S has no negative cycles, then there exists a systolic system S' equivalent to S of essentially the same structure. It be noted that equivalence is defined with regard to the input/output behaviour to a single host node to which the system is presumed to be connected.

12. Multiprocessors. In the sixties a powerful line of architectures was initiated based the connection of many full-fledged processors and memory modules into one organised scheme, with a suitable hardwired communication structure (e.g. [51]). The machines heavily use parallelism, pipelining and vector-processing. Flynn [52] suggested the often used distinction between SIMD-machines (single instruction/multiple data streaming) and MIMD-machines (multiple instruction/multiple data streaming). The latter hold promises for truly parallel processing of a single task, but the communications overhead and interference among the processors tend to spoil part of the gains of simultaneous execution.

13. Interconnection networks. Memory of an N-processor SIMD-machine is normally divided into N banks to allow for rapid parallel access. A problem of much concern has been to determine suitable networks that provide all necessary

processor-to-bank connections (and back). A crossbar switch would do, but it needs $O(N^2)$ switches.

Theorem. Any network that realizes all connections between N processors and N banks must have $\Omega(N \log N)$ switches.

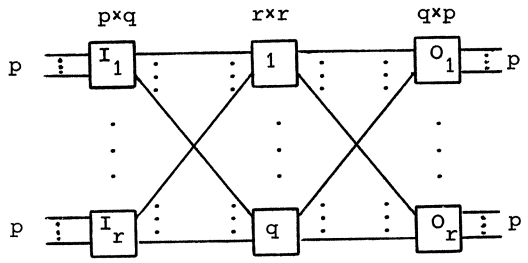
Proof

The network must be able to distinguish $\geq N!$ internal settings. If it has s switches that can be in (say) 2 states each, it can have at most 2^s different states. Thus $2^s \geq N!$, and $s \geq \log N! = \Omega(N \log N)$. □

Every network requires a routing algorithm for directing signals through the net from source to destination. We only present some results for routable and fully rearrangeable networks (see also [53]).

A useful mapping to start from is the perfect shuffle function s with $s : \{0,\ldots,N-1\} \to \{0,\ldots,N-1\}$ defined by $s(i_M i_{M-1} \cdots i_0) = i_{M-1} \cdots i_0 i_M$ when phrased in terms of binary number notation. An omega-network (Lawrie [54]) consists of log N (identical) s-stages, with lines at every level leading pairwise into N/2 switches that pass the data on or "exchange" it on the outgoing pair of lines. Effectively, a switch either applies the identity mapping or the exchange E defined by $E(i_M \cdots i_0) = i_M \cdots i_1 \bar{i}_0$. Routing from i to j is easy: slide a window over the binary expression $\lfloor i_M \cdots i_0 \overrightarrow{]} j_M \cdots j_0$ and "shuffle-exchange" i into j. Unfortunately the omega-network is not rearrangeable (in fact it isn't even non-blocking) but it can route many useful permutations correctly according to this algorithm. Parker [55] gives a neat proof that 3log N s-stages (thus, 3 passes through an omega-network) are sufficient to be able to route every permutation.

The study of rearrangeable networks has a long history in telephone systems. Let N = p.r. A starting point for much of the theory is the analysis of the "three stage" Clos networks C(p,q,r) defined as follows (each box is a suitable crossbar switch):



Theorem (Slepian-Duguid). A Clos network C(p,q,r) is rearrangeable if and only if q≥p.

Proof

Necessity of q≥p is clear, or else routings would block already in the first stage. Let q≥p and let $\Pi : \{1,\ldots,N\} \to \{1,\ldots,N\}$ be an arbitrary permutation. For p=1 it is trivial to realize Π: route all N incoming messages to the first intermediate box (which indeed has N=r inlets) and spread them according to Π from here in the second stage. For p>1 let $K_i = \{j \mid \Pi(k) \in O_j$ for some $k \in I_i\}$ be the set of indices of out-boxes that must be reached from I_i. Consider any collection $\{K_{i_m}\}_{1\leq m\leq s}$. As the s.p elements of $\bigcup_1^s I_{i_m}$ are mapped to as many outputs and each outbox can route at most p elements, they must jointly lead into ≥s outboxes. Hence $|\bigcup_1^s K_{i_m}| \geq s$, which is Hall's condition ([56]) for the existence of a set of distinct representatives. Let j_m be the representative of K_m ($1\leq m\leq r$) and k_m an input of I_m with $\Pi(k_m) \in O_{j_m}$. Route every k_m to the first immediate box and switch them there into the right permuted order. Fixing this assignment we are left to route the remaining pairs, which can be handled as if we had a C(p-1,q-1,r) net. This completes the argument by induction. □

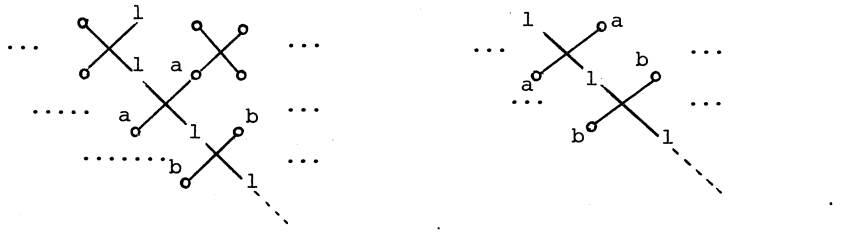
Note that we haven't used that each box is a crossbar, but merely that it is rearrangeable. This allows a recursive construction of rearrangeable networks. In particular, a Benes network B(n) is a Clos network $C(2,2,2^{n-1})$ with two B(n-1)'s as intermediate crossbars. The network has size O(N log N) for $N=2^n$. The routing problem has been studied in e.g. [57]. Many other networks are reviewed in [53].

14. Multiprocessor ("parallel") algorithms. Networks clearly are important for connecting processors themselves, but the objectives for the nets are slightly different. There is a need for fast exchange of information and broadcasting of signals. Stone [58] has shown that N processors connected in a perfect shuffle allow for extremely efficient execution of several standard algorithms (e.g. polynomial evaluation, sorting, the FFT). In some formulations processors are also paired in blocks of two, leading to the pattern of the shuffle-exchange graph. At some level of abstraction algorithm design could specify both the processor tasks and the assumed interconnection pattern, leaving the scheduling on the actual multiprocessor for a second "pass". See e.g. [48].

Theorem. A linear array of N suitably instructed processors can sort N numbers in O(N) time.

Proof

The method is based on odd-even transposition sorting. Put N keys in N processors, numbered odd and even alternately. Assume the even processors are activated first. In each cycle the following takes place: the key in every activated processor is compared with the key in its right neighbor and exchanged when the latter is smaller. Odd and even processors are activated alternately. One can prove by induction that the algorithm sorts the keys within N cycles. Note that the largest key 1 always wins and moves across to the right (it hesitates in the first cycle if it is stored in an odd processor):



Its path separates the computation in two triangular areas. Imagine 1 is dropped. Then the right upper computation can be moved down and left one

step, to merge with the left lower past to an odd-even transposition sort on N-1 keys. To top row left of 1 does no harm and can be excluded for its effect. By induction N-1 cycles do the job in the remaining sort, hence N are sufficient for the original set of keys. □

Baudet and Stevenson [59] have investigated the effect of giving each processor some memory to hold a sorted subsequence rather than just a single key. The comparison-exchange operation becomes a merge-split operation on sequences. They show that N keys may be sorted on an array of p processors in $O(\frac{N}{p} \log \frac{N}{p} + N)$ time, provided that each processor can hold $\frac{N}{p}$ keys. The sorting problem was addressed also in e.g. [60] and [61], where a feasible algorithm was proposed to sort N keys in O(N) time by log N processors, one corresponding to each level of the familiar merge-sort routine. Batcher's bitonic sort method needs only $O(\log^2 N)$ time but uses N processors with a very specific interconnection pattern (see [58]).

In a variety of studies detailed considerations about processor structure and interconnection have been de-emphasized. Processors are assumed available in unlimited quantity, with any form of desired signaling (broadcasting) and shared memory. A rule is required to resolve conflicts in simultaneous reads or writes of a same memory location. The assumption of unlimited processors can be justified from a simple observation due to Brent [62]:

Theorem. If a computation can be performed in time t with sufficiently many processors that perform q operations total (with each operation requiring one time unit), then the computation can be performed in time $t + (q-t)/p$ with p such processors.

Proof

Suppose s_i operations are performed in parallel during step i ($1 \leq i \leq t$), with $q = \sum_1^t s_i$. Using p processors we can stimulate step i in time $\lceil s_i/p \rceil$. The entire computation is thus rescheduled and requires a number of steps of about $\sum_1^t \lceil s_i/p \rceil \leq \sum_1^t (s_i+p-1)/p = (1-\frac{1}{p})t + \frac{1}{p}.\sum_1^t s_i = t + (q-t)/p$. □

As an example of a computation with unbounded parallelism and simultaneous memory access, we note the following result of Kučera [63].

Theorem. The minimum of N keys can be computed in O(1) time using N^2 processors, allowing "weak" memory conflicts.

Proof

Let the keys be stored in a[1] to a[N] and use additional (shared) locations b[1] to b[N]. The processors P_{ij} ($1 \leq i,j \leq N$) execute the following 4 cycles. In cycle 1 every P_{i1} ($1 \leq i \leq N$) writes 0 into b[i] for initialization and the other processors are silent. In cycle 2 the P_{ij} write 1 into b[j] if a[i]<a[j]. As a result, b[i] = 0 iff a[i] = min{a[1],...,a[N]}. To find (say) the smallest index of a minimal key, cycle 3 lets the P_{ij} write 1 into b[j] when i<j and b[i] = 0. In cycle 4 P_{ij} ($1 \leq i \leq N$) inspects b[i] and outputs a[i] as smallest key when its value is 0. □

Parallel computation of ranks was studied (with different assumptions on memory use) in e.g. [64] and [65].

The use of many processors distorts the view of the actual computational gains over a single processor. Let T_p ($p \geq 1$) be the computation time for a problem using p processors. The "speedup" of a parallel algorithm may be defined as $S_p = T_1/T_p$. (Using Brent's theorem it follows that $S_p \leq p$.) The efficiency of a parallel algorithm can be defined as $E_p = S_p/p$ ($= T_1/p.T_p$). The Amdahl effect ([66]) asserts that for many practical architectures the efficiency does not rise with an increased number of processors, for reasons of housekeeping and communications chores and the lack of a sufficient and regular form of parallelism in the problem solved.

Techniques for designing parallel algorithms include (i) recursive doubling, (ii) broadcasting, (iii) decomposition into weakly dependent parts and (iv) simultaneous building.

Theorem. $\sum_{0}^{N-1} a_i$ (and $\prod_{0}^{N-1} a_i$) can be computed in O(log N) time, using N/2 processors.

Proof

Use the processors to compute $a_{2i} + a_{2i+1}$ ($0 \leq i \leq \frac{N}{2} - 1$) in the first step. Recursively double the extent of the partial sums using $\frac{1}{4}N$, $\frac{1}{8}N$, ... of the processors until we have $a_0 + \ldots + a_{\frac{N}{2}-1}$ and $a_{\frac{N}{2}} + \ldots + a_{N-1}$ and one proces-

sor can compute the final result in one more step. (We assumed that $N=2^r$, some r.) □

Compared to the sequential algorithm for $\sum_0^{N-1} a_i$ recursive doubling gives a speedup of O(N/log N) but an efficiency of only O(1/log N). Two N×N matrices can be multiplied in O(log N) time as well, using N^3 processors. Use a cluster of N processors for each of the N^2 elements of the product matrix. In one step they compute (multiply) the summands, and O(log N) steps of recursive doubling suffice to accumulate the sums. A few years ago Csanky [67] proved that N×N matrices can be inverted in $O(\log^2 N)$ time, still using a polynomial number of processors.

The idea of broadcasting is illustrated in the parallel solution of a linear system x = Ax+b ($x \in \mathbb{R}^N$) with A lower triangular. All multiterm linear recurrences can be put in this form. Clearly $x_1=b_1$ and the straight computation of $x_2, x_3, \ldots$ would take about N^2 steps on a single processor.

Theorem. A linear system x = Ax+b with A an N×N lower triangular matrix can be solved in O(N) time, using N-1 processors.

Proof

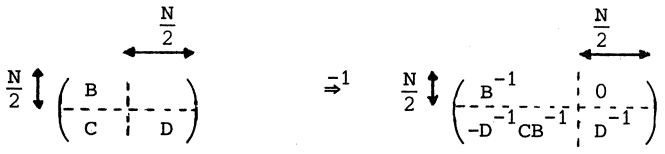
Use processors P_i ($2 \leq i \leq N$). After eliminating x_{j-1} ($j \geq 2$) assume that the P_i with $i \geq j$ have $a_{i1}x_1 + \ldots + a_{ij-1}x_{j-1}$ in store. In the next cycle P_j can compute x_j. It subsequently broadcasts the value to all P_i with i>j, which compute $a_{ij}.x_j$ and add it to the partial sum they hold. □

The algorithm, known as the "column sweep method", yields a speed-up of O(N) and an efficiency of about $N^2/(N-1)\times 2N \geq \frac{1}{2}$ (a constant, anyway). By increasing the number of processors one can lower the time bound to $O(\log^2 N)$, as one might expect from Csanky's result. The simple proof though illustrates another useful technique (from [68]).

Theorem. A linear system x = Ax+b with A an N×N lower trangular matrix can be solved in $O(\log^2 N)$ time, using $O(N^3)$ processors.

Proof

As $x = (I-A)^{-1}b$ we are done if we prove that a lower triangular matrix can be inverted within the bounds stated. Decompose (split) the matrix as



, with B and D lower triangular, and observe the (recursive) structure of the inverse. If B^{-1} and D^{-1} are found, only $O(\log N)$ steps and $O(N^3)$ processors are required to "finalize" A^{-1}. Alltogether this yields an algorithm of the desired complexity. □

Schendel [68] gives a readable account of "parallel (numerical) mathematics".

An example of simultaneous building is provided by Sollin's algorithm for determining a minimum spanning tree of a graph. The graph is given by an adjacency matrix, which lists the weight of edges $\overline{v_i v_j}$ in entry (i,j) (with ∞ denoting the absence of an edge).

Theorem. A minimum spanning tree of a weighted N-node graph can be computed in $O(\log N)$ time, using $O(N^3)$ processors.

Proof

Sollin's algorithm relies on maintaining a global invariant that at any stage the disjoint subtrees obtained are subtrees of one minimum spanning tree. Use N processors P_i, with P_i corresponding to v_i ($1 \leq i \leq N$). Each P_i will hold some label uniquely identifying the tree to which v_i currently belongs. The algorithm does the following:

(i) in a first cycle, each P_i determines a lightest edge $\overline{v_i v_j}$ incident to v_i ($1 \leq i \leq N$). To prevent cycles, the lightest edge with minimum j is taken. The computation, and subsequent administration, needs only $O(1)$ time if sufficiently many auxiliary processors are on hand.

(ii) as long as there still are >1 subtrees, do the following next cycle. For each subtree T_l determine a lightest edge $\overline{v_i v_j}$ connecting to a different T_m. To prevent cycles again, the edge with lexicographically smallest i,j is

taken. T_l and T_m are subsequently connected and relabeled. The step is more involved, but can be done in O(1) time with auxiliary processors.

In each cycle the number of disjoint subtrees is halved, and the algorithm terminates after log N cycles of O(1) time each.

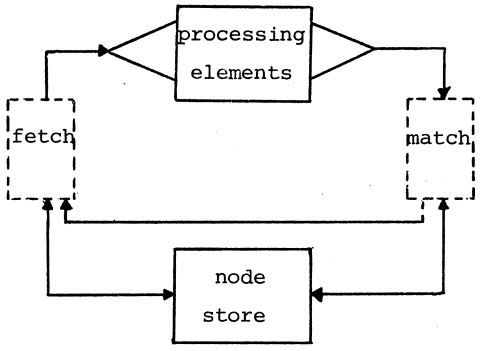
The timing of Sollin's algorithm is different depending on the memory model used (we allowed "unambiguous" access conflicts). Bentley and Ottmann [70] proved that the algorithm can run in O(N log N) time on a linear array of N processors.

15. Dataflow computing. The earlier development suggests an entirely different approach to programming, based on clusters (nodes) and information transfer through communication lines rather than through variables in some memory. At the level of individual instructions this leads to Dennis' dataflow concept ([71]) which holds the view that an instruction is ready for execution when its operands are available. To support and implement this concept a very different form of computer is required to realize the intrinsic parallelism of execution for many simultaneously active instructions. The idea of data-driven computation itself is not new, but only in recent years have architectural schemes with an attractive expected performance been developed and in some cases experimented with (see [72] for an extensive survey).

In its most primitive form, a data flow program is a directed graph in which the nodes represent processing elements of some sort and the edges represent datapaths. There is no global memory and (hence) there are no variables, but data (tokens) is transmitted directly from node to node over existing datapaths. Processing elements digest tokens from their incoming edges and emit new tokens over their outgoing edges, presumably after some internally specified computation. The execution of one "cycle" is very similar to a firing in the terminology of Petri-nets. Processing elements are operators, i.e., fixed token-mappings of some variety. Except that cycles and token-transports take finite time, no further assumptions are made about the speeds or relative speeds of the processing elements or when processing elements choose to take in a next batch of input. Dataflow computation is completely asynchronous. As a consequence, tokens may have to queue along a datapath

if the node "at the other end" is not processing fast enough. A processing element must "wait" whenever it wants a token from an empty input line or some rule prevents it from further sending on a congested output-line. Jaffe [73] and Böhm and van Leeuwen [74] present approaches for a fundamental analysis of the underlying computational model. Algorithms can be designed in dataflow that achieve the tight bounds of many known multiprocessor algorithms, without the need for global control (see [75]).

Dataflow machines are designed for rapid execution of dataflow programs. The basic instruction execution mechanism used in virtually all machines is the circular pipeline or "ring":



Using program information from the node store, the fetch unit assembles activated instructions to tokens and feeds them to a pool of processors. Result tokens are received by the match unit which checks, according to some policy, what instructions now have a fully set of operands. Any one that has is queued to the fetch unit. Ultimately, the level of concurrency achieved by an architecture of this type is limited by the capacity of the datapaths in the ring. Nevertheless, it is a radical departure from the classical "von Neumann" architecture and a bold attempt to exploit concurrency of computation truly and at a large scale. Dennis [75] describes a possible extension of the approach to dataflow multiprocessors. Several languages (see e.g. [76]) have been designed to support dataflow programming.

16. Models of parallel computation. Models of computation enable one to analyse and prove fundamental results about the power and limitations of a real or proposed machine architecture. As modern technology is moving towards highly integrated circuitry and novel architectures, we need to revise our ideas about computation and the way it is performed accordingly. As we have seen, there appears to be a distinction between models based on a fixed connection network of processors and models based on the existence of global or shared memory. In the former category there are linear-, mesh- and tree-connected arrangements of processors. Wittie [77] surveys many other patterns that are of some practical importance. Galil and Paul [78] have taken a broad view and modeled a parallel machine as an infinite recursive graph, with some recursive assignment of nodes to processors. The processors may be finite automata, RAM's or limited RAM's of some sort and at every step each processor consults the processors on adjacent nodes before going through its compute cycle. Every determinsitic multitape Turing machine with time bound T can be simulated by a tree-connected parallel machine of finite automata in $O(T \log \log T/\log T)$ time. Many other complexity questions are explored. See also [79], [80]. Alternation has been another fundamental notion (e.g. [81]) that proved useful in clarifying the connections between sequential and parallel time and space measures.

Fortune and Wyllie [82] proposed a very general and flexible model of parallel computation (the P-RAM) based on random access machines that operate in parallel. The machines have unbounded local memory but can communicate only through a shared (and unbounded) global memory. Simultaneous reads of a location are allowed, but simultaneous writes block the P-RAM. The random access machines act synchronized, executing one instruction (in parallel) per time unit. The most powerful instruction is the FORK, which enables a processor to activate a next free processor and start it off at some entry point of the parallel program. It is shown that deterministic P-RAMs can accept in polynomial time precisely the sets accepted by (sequential) Turing machines in polynomial space. Nondeterministic P-RAMs accept in polynomial time precisely the sets by nondeterministic Turing machines in exponential time.

17. Buses. Processors and memory modules of different sorts and uses may be tied into one system, as in the machine room of a computation centre. It is usually done to off-load the central processor and to provide for access to specialized devices or back-up store. The communication between the differen processors is usually realized by a transport circuit, called a "bus". Buses differ by the speed and form of transport (bit-serial or bit-parallel). Also, all processors connected to the bus see the same signals. It is therefore important that some discipline is enforced (called the bus access protocol) for conflict-free, yet expedient sharing of the bus. There are two approaches to the sharing problem. One is to have a central bus controller, which polls processors and schedules bus use. Another is to distribute control and implement a suitable protocol in every processor. When several processors try to write on the bus contention occurs. It is usually detected by hardware means, and solved by some form of recovery and a retry. In the case of transports over longer distances, the propagation delay of signals over the bus cannot be neglected and becomes a factor in deciding an efficient multiplexing or sharing algorithm.

18. Remote access. The use of terminals has been a first step to distribute a system over a wider area. Terminals connect to a shared (multiplexed) port of the central computer or to a local concentrator, which attempts to optimize the transports to and from the central site. Terminal handling has contributed to much of the essential understandings about data communication (see e.g. [83]) on the one hand, and parallel processing of jobs on the other.

19. Computer networking. To access different sites from one host, networks have been designed of ever increasing complexity. The reason is usually the desire to communicate information, access data or use some specialized facility. There are two essential principles for realizing computer-to-computer communication: (i) circuit switching, (ii) message switching. Tanenbaum [84] gives a detailed description and explains many more of the problems in transfering data from one computer to another. Martin [85] gives a good overall account of the objectives of computer networks.

20. Distributed processing. As the potential of computer networks was recognized, it became an end in itself to provide all the required facilities for users somewhere on the network. It gives the luxury of a large system at shared expenses. Bochmann [86] recognizes the following three principles of distributed processing: (i) processing can be done where the data is, (ii) redundancy (back-up if one processing unit goes down) and (iii) economy (dedicated units need not be available everywhere). Local system versus communication costs will determine the optimal topology and policies of the network.

21. Local area networks. Networks have different policies depending on their scale (and, of course, their architecture). Local area networks allow for high-speed transmissions at a very low error rate. Given these considerations, local area networks can afford to have a simple topology (one node will never be far away from another) which, again, keeps the added communications overhead for routing very small. One wellknown design is Ethernet ([87]), which has the properties of a contention bus. An Ethernet consists of a single (or split) co-ax cable with taps that provide the points to which processors can be connected. A processor only transmits when the Ethernet appears quiet. Its packets essentially make a round trip over the cable and (thus) certainly pass their intended destination. It is left to the individual stations to recognize and intercept the packets for their use. The Ethernet thus effectively realizes a broadcast medium (or "ether"). While transmitting a processor listens whether another processor has perhaps begun transmitting too, in which case an "audible" collision occurs. If so, both processors immediately stop transmitting, pause a random period and try to transmit again. A processor can never be sure its packet reached its destination without interference until after the time for a full round trip (only a few microseconds!). The Ethernet is engineered on the assumption that collisions happen rarely. It is an example of a CSMA ("Carrier Sense Multiple Access") network. Greenberg [88] has given an interesting analysis of the expected time of some simple distributed control tasks over an Ethernet-like medium.

Rings are another topology applied in local area networks. The operation of a ring network hinges upon three main ingredients: (i) the transmission policy used by nodes to place packets on the ring, (ii) the reception policy

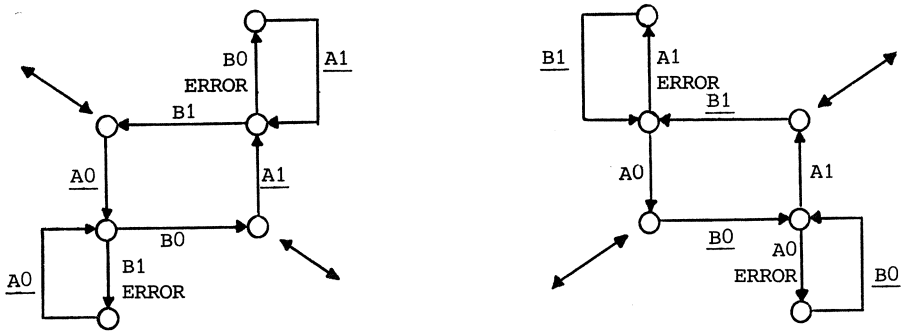
used to decide if a packet is to be received, and (iii) the packet erase policy (to use in case a packet appears to circulate indefinitely). There are three major types of ring architectures in use, which differ largely by the transmission policy that is used: (i) slotted rings, (ii) token rings and (iii) insertion rings. See [84] or [89] for further details.

22. Protocols. A network consists of a collection of interconnected processors (nodes) that exchange data and messages over some nontrivial distance. The orderly exchange of information requires that the nodes conform to some pre-established agreements or rules which constitute a protocol. A protocol specifies both the format of the information packages transmitted and the actions to be taken for sending and receiving, as the communication ("control") between the nodes to set up or maintain a connection. A protocol thus embodies all the necessary actions to let the network function. To incorporate it, the network carries both data and control messages, in separate or combined packets. Most networks use some form of send/acknowledge protocol to set up connections, to check packet arrival and/or the error-freeness of a transmission. In case an error occurred (e.g. by parity control) the packet must be send again. For straight point-to-point connections over a half-duplex line there is a classical observation of Bartlett, Scantlebury and Wilkinson [90]:

Theorem. One bit suffices for error control when transmitting over a half-duplex line.

Proof

The technique is known as the "alternating bit" protocol. Imagine two hosts A and B communicating over a half-duplex line, taking turns in sending data and control information. Let A0 (and so on) mean that A sends a packet with control bit 0. Let A0 (and so on, not underlined) mean that a packet from A with control bit 0 is received (by B). The protocol for A and B is best described by the following two communicating finite state diagrams:



A B

The diagrams should be read thus that when e.g. B receives a damaged packet with control bit 0, then it sends a control bit 1 back to A and A responds by transmitting the same packet with control bit 0 again. □

Clearly more involved protocols are needed (and have been designed!) in networks where nodes would waste time for acknowledgements of every separate packet and like to use the medium continuously. This opens the way for extremely complicated communications with control messages, packets and retransmitted packets in one stream that should eventually carry all data to correct arrival. Sunshine [91] gives an account of the problems in this direction. Protocol validation has been attacked by methods derived from the correctness theory of parallel programs (e.g. [92]).

23. Routing. In a packet-switching network some strategy is required for directing packets from source to destination through the transmission medium. An optimal strategy should deliver a largest possible number of packets in a shortest possible time. Packets of a message are sent ("hop") from node to node to reach their destination, but need not all follow the same path. Thus sequence numbers are needed and the receiving host may have some difficulties in assembling the message that is coming in. The routing algorithm of the network must avoid congestion of the imp's on the net and be proof against failures of some parts of the net. All routing algorithms are based

on maintaining routing tables either at a central node or distributed over all nodes. The routing tables contain information about connections, distances and delays to be expected along various lines. By now there is an extensive and non-trivial litterature concerning non-adaptive and adaptive routing (see e.g. [93]). Santoro and Khatib [94] show that in simple cases no routing tables are needed.

The design of network wide systems leads to many problems about distributed algorithms and computing that can now be envisaged. Timing (and the notion of time itself), event ordering, synchronization, data integrity, encryption and compression are only the beginning of an endless list of issues that can be brought to bear on distributed computing in this wide sense. A unique account of the design and inplementation problems for distributed systems is given in [1].

References.

[1] D.W. Davies et.al., *Distributed systems - architecture and implementation*, LN-CS vol. 105, Springer Verlag, Heidelberg, 1981.

[2] J.L. Peterson, *Petri nets*, Comp. Surv. 9 (1977) 223-252.

[3] M. Hack, *The recursive equivalence of the reachability problem and the liveness problem for Petri nets and vector addition systems*, Project MAC, Memo 107, MIT, Cambridge, Mass., 1974.

[4] E.W. Mayr, *An algorithm for the general Petri net reachability problem*, Proc. 13th Ann. ACM Symposium on Theory of Computing, pp. 238-246, 1981.

[5] S.R. Kosaraju, *Decidability of reachability in vector addition systems*, Proc. 14th Ann. ACM Symposium on Theory of Computing, pp. 267-281, 1982.

[6] G. Kahn, *The semantics of a simple language for parallel programming*, in: J. Rosenfeld (ed.), IFIP 74, North-Holland Publ. Comp., Amsterdam, pp. 471-475, 1974.

[7] C. Hewitt and H. Baker, *Laws for communicating parallel processes*, in: B. Gilchrist (ed.), IFIP 77, North-Holland Publ. Comp., Amsterdam, pp. 987-992, 1977.

[8] R. Milner, *A calculus of communicating systems*, LN-CS vol. 92, Springer Verlag, Heidelberg, 1980.

[9] V.R. Pratt, *Process logic*, Proc. 6th Ann. ACM Symposium on Principles of Programming Languages, pp. 93-100, 1979.

[10] J.W. de Bakker and J.I. Zucker, *Denotational semantics of concurrency*, Proc. 14th Ann. ACM Symposium on Theory of Computing, pp. 153-158, 1982.

[11] M. Nivat, *On the synchronization of processes*, Rapp. No. 3, INRIA, Rocquencourt, 1980.

[12] V.R. Pratt, *On the composition of processes*, Proc. 9th Ann. ACM Symposium on Principles of Programming Languages, pp. 213-223, 1982.

[13] E.W. Dijkstra, *Cooperating sequential processes*, in: F. Genuys (ed.), Programming Languages, Acad. Press, New York, pp. 43-112, 1968.

[14] N. Wirth, *A note on "Program structures for parallel processing"*, CACM 9 (1966) 320-321.

[15] E.W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, CACM 18 (1975) 453-458.

[16] P. Brinch Hansen, *Structured multiprogramming*, CACM 15 (1972) 574-578.

[17] C.A.R. Hoare, *Towards a theory of parallel programming*, Int. Sem. on Operating System Techniques, Belfast, 1971 (see also: P. Brinch Hansen, Operating system principles, Prentice Hall, Englewood Cliffs, NJ, 1973).

[18] C.A.R. Hoare, *Monitors: an operating system structuring concept*, CACM 17 (1974) 549-557.

[19] N. Wirth, *Modula: a language for modular multiprogramming*, Software: Pract. & Exper. 7 (1977) 3-35.

[20] P. Brinch Hansen, *The architecture of concurrent programs*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[21] J. Welsh and D.W. Bustard, *Pascal-Plus: another language for modular multiprogramming*, Software: Pract. & Exper. 9 (1979) 947-958.

[22] P. Brinch Hansen, *Distributed processes: a concurrent programming concept*, CACM 21 (1978) 934-941.

[23] C.A.R. Hoare, *Communicating sequential processes*, CACM 21 (1978) 666-667.

[24] N. Wirth, *MODULA-2*, Rep. 36, Institut f. Informatik, ETH, Zürich, 1980.

[25] J.G.P. Barnes, *Programming in ADA*, Addison Wesley Publ. Comp., London, 1982.

[26] C.A.R. Hoare, *An axiomatic basis for computer programming*, CACM 12 (1969) 576-583.

[27] S. Owicki, *Axiomatic proof techniques for parallel programs*, TR-75-251, Dept. of Computer Science, Cornell University, Ithaca, NY, 1975.

[28] D. Gries, *An exercise in proving parallel programs correct*, in: Language hierarchies and interfaces, LN-CS vol. 46, Springer Verlag, Heidelberg, pp. 57-81, 1976.

[29] K.R. Apt, N. Francez and W.P. de Roever, *A proof system for Communicating sequential processes*, ACM Trans. Progr. Lang. & Syst. 2 (1980) 359-385.

[30] R.T. Gerth, *A sound and complete Hoare axiomatization of the ADA rendezvous*, Techn. Rep. RUU-CS-82-5, Dept. of Computer Science, University of Utrecht, Utrecht, 1982.

[31] G. Lelann, *Motivations, objectives and characterizations of distributed systems*, in: [1], pp. 1-9.

[32] C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wesley Publ. Comp., Reading, Mass., 1980.

[33] C.D. Thompson, *A complexity theory for VLSI*, Techn. Rep. CMU-CS-80-140, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1980.

[34] F.T. Leighton, *New lower bound techniques for VLSI*, Proc. 22nd Ann. IEEE Symposium on Found. of Computer Science, pp. 1-12, 1981.

[35] F. Harary, *Graph theory*, Addison Wesley Publ. Comp., Reading, Mass., 1969.

[36] R.J. Lipton and R.E. Tarjan, *A separator theorem for planar graphs*, SIAM J. Appl. Math. 36 (1979) 177-189.

[37] J. Vuillemin, *A combinatorial limit to the computing power of VLSI circuits*, Proc. 21st Ann. IEEE Symposium on Found. of Computer Science, pp. 294-300, 1980.

[38] R.P. Brent and H.T. Kung, *The chip complexity of binary arithmetic*, Proc. 12th Ann. ACM Symposium on Theory of Computing, pp. 190-200, 1980.

[39] H.T. Kung, B. Sproull and G. Steele (eds.), *VLSI systems and computations*, Proc. CMU Conf., Computer Science Press, Rockville, Md., 1982.

[40] M.R. Kramer and J. van Leeuwen, *Wire-routing is NP-complete*, Techn. Rep. RUU-CS-82-4, Dept. of Computer Science, University of Utrecht, Utrecht, 1982.

[41] M.R. Kramer and J. van Leeuwen, *The NP-completeness of finding minimum area layouts for VLSI-circuits*, Techn. Rep. RUU-CS-82-6, Dept. of Computer Science, University of Utrecht, Utrecht, 1982.

[42] H. Bodlaender, M. Kramer, J. van Leeuwen, M.H. Overmars, A.A. Schoone, R. Tan and H. Wijshoff, *Plane realisation of 3-dimensional VLSI-designs*, to appear.

[43] B. Chazelle and L. Monier, *A model of computation for VLSI with related complexity results*, Techn. Rep. CMU-CS-81-107, Dept. of Computer Sci., Carnegie-Mellon University, Pittsburgh, 1981.

[44] B. Chazelle and L. Monier, *Unbounded hardware is equivalent to deterministic Turing machines*, Techn. Rep. CMU-CS-81-143, Dept. of Computer Sci., Carnegie-Mellon University, Pittsburgh, 1979.

[45] H.T. Kung, *Let's design algorithms for VLSI systems*, Techn. Rep. CMU-CS-79-151, Dept. of Computer Sci., Carnegie-Mellon University, Pittsburgh, 1979.

[46] M.J. Foster and H.T. Kung, *Design of special purpose VLSI chips: examples and opinions*, Techn. Rep. CMU-CS-79-147, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, 1979.

[47] H.T. Kung and C.E. Leiserson, *Systolic arrays for (VLSI)*, Techn. Rep. CMU-CS-79-103, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1979.

[48] H.T. Kung, *The structure of parallel algorithms*, Techn. Rep. CMU-CS-79-143, Dept. of Computer Sci., Carnegie-Mellon University, Pittsburgh, 1979.

[49] H.M. Ahmed, J.M. Delosme and M. Dorf, *Highly concurrent computing structures for matrix arithmetic and signal processing*, Computer (1982) 65-82.

[50] C.E. Leiserson and J.B. Saxe, *Optimizing synchronous systems*, Techn. Rep. CMU-CS-82-101, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1982.

[51] G.H. Barnes et. al., *The ILLIAC IV computer*, IEEE Trans. Comput. C-17 (1968) 746-757.

[52] M.J. Flynn, *Some computer organisations and their effectiveness*, IEEE Trans. Comput. C-21 (1972) 948-960.

[53] C.D. Thompson, *Generalized connection networks for parallel processor intercommunication*, IEEE Trans. Comput. C-27 (1978) 1119-1125.

[54] D.H. Lawrie, *Access and alignment of data in an array processor*, IEEE Trans. Comput. C-24 (1975) 1145-1155.

[55] D.S. Parker jr., *Notes on shuffle/exchange - type switching networks*, IEEE Trans. Comput. C-29 (1980) 213-222.

[56] V.E. Benes, *Mathematical theory of connecting networks and telephone traffic*, Acad. Press, New York, 1965.

[57] D. Nassimi and S. Sahni, *A self-routing Benes network and parallel permutation algorithms*, IEEE Trans. Comput. C-30 (1981) 332-340.

[58] H.S. Stone, *Parallel processing with the perfect shuffle*, IEEE Trans. Comput. C-20 (1971) 153-161.

[59] G. Baudet and D. Stevenson, *Optimal sorting algorithms for parallel computers*, IEEE Trans. Comput. C-27 (1978) 84-87.

[60] F.P. Preparata, *New parallel-sorting schemes*, IEEE Trans. Comput. C-27 (1978) 669-673.

[61] S. Todd, *Algorithm and hardware for a merge sort using multiple, processors*, IBM J. Res. Develop. 22 (1978) 509-517.

[62] R.P. Brent, *The parallel evaluation of general arithmetic expressions* JACM 21 (1974) 201-206.

[63] L. Kučera, *Parallel computation and conflicts in memory access*, Inf. Proc. Lett. 14 (1982) 93-96.

[64] L.G. Valiant, *Parallelism in comparison problems*, SIAM J. Comput. 4 (1975) 348-355.

[65] Y. Shiloach and U. Vishkin, *Finding the maximum, merging and sorting in a parallel computation model*, J. Algor. 2 (1981) 88-102.

[66] G.M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proc. Spring Joint Computer Conf., pp. 483-485, 1967.

[67] L. Csanky, *Fast parallel matrix inversion algorithms*, SIAM J. Comput. 5 (1976) 618-623.

[68] D. Heller, *A survey of parallel algorithms in numerical linear algebra*, Techn. Rep., Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1976.

[69] U. Schendel, *Einführung in die parallele Numerik*, Oldenbourg Verlag, München, 1981.

[70] J. Bentley and Th. Ottmann, *The power of a one-dimensional vector of processors*, Bericht 89, Inst. f. Angew. Informatik u. formale Beschreibungsverf., Univ. Karlsruhe, Karlsruhe, 1980.

[71] J.B. Dennis, *First version of a dataflow procedure language*, in: Programming Symposium, LN-CS vol. 19, Springer Verlag, Heidelberg, 1974, pp. 362-376.

[72] P.C. Treleaven, D.R. Brownbridge and R.P. Hopkins, *Data-driven and demand-driven computer architecture*, Comp. Surv. 14 (1982) 93-143.

[73] J.M. Jaffe, *The equivalence of r.e. program schemes and dataflow schemes*, J. Comput. Syst. Sci. 21 (1980) 92-109.

[74] A.P.W. Böhm and J. van Leeuwen, *A basis for dataflow computing*, Techn. Rep. RUU-CS-81-6. Dept. of Computer Science, University of Utrecht, Utrecht, 1981.

[75] J.B. Dennis, *Data flow supercomputers*, Computer (1980) 48-56.

[76] W.B. Ackerman and J.B. Dennis, *VAL: a value oriented algorithmic language* (prelim. ref. manual), TR-218, Lab. for Computer Sci., MIT, Cambridge, Mass., 1979.

[77] L.D. Wittie, *Communication structures for large networks of micro-computers*, IEEE Trans. Comp. C-29 (1980).

[78] Z. Galil and W.J. Paul, *A theory of complexity of parallel computation*, preprint, 1980 (also: An efficient general purpose parallel computer, Proc. 13th Ann. ACM Symposium on Theory of Computing, pp. 247-256, 1981).

[79] F. Meyer auf der Heide, *Time-processor trade-offs for universal parallel computers*, preprint, Fac. of Mathematics, Univ. Bielefeld, Bielefeld, 1981.

[80] F. Meyer auf der Heide, *Efficiency of universal parallel computers*, Int. Bericht 1/82, Fachber. Informatik, Univ. Frankfurt, Frankfurt, 1982.

[81] A. Chandra and L. Stockmeyer, *Alternation*, Proc. 17th Ann. IEEE Symposium on Found. of Computer Science, pp. 98-108, 1976.

[82] S. Fortune and J. Wyllie, *Parallelism in random access machines*, Proc. 10th Ann. ACM Symposium on Theory of Computing, pp. 114-118, 1978.

[83] J.E. McNamara, *Technical aspects of data communication*, 2nd ed., Digital Press, Bedford, Mass., 1982.

[84] A.S. Tanenbaum, *Computer networks*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[85] J. Martin, *Computer networks and distributed processing: software, techniques and architecture*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[86] G. v. Bochmann, *Architecture of ditributed computer systems*, LN-CS vol. 77, Springer Verlag, Heidelberg, 1979.

[87] R.M. Metcalfe and D.R. Boggs, *Ethernet: distributed packet switching for local computer networks*, CACM 19 (1976) 395-404.

[88] A.G. Greenberg, *On the time complexity of broadcast communication schemes*, Proc. 14th Ann. ACM Symposium on Theory of Computing, pp. 354-364, 1982.

[89] R.P. Lee, *The architecture of a dynamically reconfigurable insertion ring network*, Rep. RJ 2485 (32434), IBM Research Lab., San Jose, Ca., 1979.

[90] K.A. Bartlett, R.A. Scantlebury and P.T. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, CACM 12 (1969) 260-261.

[91] C.A. Sunshine, *Formal techniques for protocol specification and verification*, Computer 12 (1979) 20-27.

[92] B. Hailpern and S. Owicki, *Modular verification of computer communication protocols*, Rep. RC 8726 (#38174), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1981.

[93] M. Schwartz, *Routing and flow control in data networks*, Rep. RC 8353 (#36329), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1980.

[94] N. Santoro and R. Khatib, *Routing without routing tables*, Rep. SCS-TR-6, School of Computer Science, Carleton University, Ottawa, Canada, 1982.

PARALLEL COMPUTATION

L.G. Valiant
Harvard University, Cambridge, USA

0. INTRODUCTION

There is little doubt that in the future computers will exploit parallelism much more than they do at present. There is equally little doubt that this exploitation will present formidable engineering challenges to both computer designers and programmers. In this paper I will try to advocate the view that in addition to the engineering considerations there are also some theoretical issues whose resolution is likely to contribute to future computing practice. To this end I will discuss two fundamental questions that have fascinated me for some time:

(a) Can we characterize the classes of problems that are amenable to fast parallel computation?

(b) Are there computer architectures that can exploit the inherent parallelism in all problems reasonably efficiently, or are all efficient parallel computers necessarily special purpose?

Recent work provides encouragingly positive partial solutions to these problems, and it is these that I plan to review. There are, of course, numerous issues important in parallel computation that are not covered by these two questions as here formulated. Discussion of these is omitted for the sake of brevity.

1. INHERENT PARALLELISM IN COMPUTATIONAL PROBLEMS

Given a particular computation (i.e. a run of a program on a particular input) the notion of inherent parallelism is well defined. A computation can be represented by an acyclic directed graph that represents the data dependencies among the inputs and computed values. The depth of the graph gives the minimum number of parallel time units needed to simulate the computation if at any step we can perform in parallel an arbitrary number of computation

steps on data then available. For many programming constructs (e.g. loops where the loop variable has input independent range) the computation graph is independent of the input data. Hence we can analyze the inherent parallelism in such programs fairly easily. Indeed, much empirical work has been done in determining the inherent parallelism in existing programs written for sequential execution [Kuck].

In the present paper we are not concerned with the inherent parallelism in fixed programs but with the parallelism inherent in *problems*. Given a problem specification (which may be a sequential program) does there exist any program for it with large inherent parallelism?

There are many examples of computationally trivial problems that appear difficult to parallelize. Let us consider undirected graphs with n nodes and define a *clique* to be "a complete subgraph that cannot be extended to a larger complete subgraph". For example, if the graph has an isolated node then this node forms a clique. If a node has some edge incident to it then it is not a clique since it is contained in a complete subgraph of size at least two. The problem of finding a clique of some predetermined size is difficult [C1]. Here we are concerned with the following easier problem: Given a graph find *any* clique in it. Sequentially this is trivial since we can start with any node and repeatedly augment it to larger and larger complete subgraphs in any way we like until we can make no further progress.

Suppose that we wish to find such a clique in parallel. Although we have not defined any model of computation the reader will appreciate where the logical problem lies. The sequential algorithm has up to n stages and it is difficult to see how the problem can be solved in substantially fewer stages, such as $O(\sqrt{n})$ or $O(\log n)$.

There are numerous such combinatorial problems for which easy sequential algorithms exist but no way of parallelizing is known. Unfortunately for none of these problems do we have any proofs to show that parallelizing is impossible. The only guidance we have is that for certain specific problems we can prove that they are as difficult to parallelize as any other. Hence they are generally conjectured to be unparallelizable since otherwise all problems would be parallelizable and this is contrary to current belief. The notion of a problem being as difficult as any is formalized as "complete for polynomial time Turing computations under logarithmic space reducibility" by COOK [C2] and as "complete for polynomial size circuits under p-projections" by the author [V1]. A typical example of such a problem is linear programming [DLR,V1].

1.1. Polynomials.

Algebraic complexity theory provides a very clean context and model for understanding parallelism. Here we seek to compute a polynomial $P[x_1,\ldots,x_n]$ in indeterminates $x_1,\ldots,x_n$ with coefficients from a field F such as the real numbers. The model of computation is that of straight line programs that allow constants from F and indeterminates as inputs and $\{+,\times\}$ as operations [BM]. This is a powerful model. For example, subtraction can be simulated by addition using the constant "-1". More surprisingly division can be simulated fairly efficiently within it under very general conditions [Str,BGH].

Starting with the simplest polynomial x^n, it is clear that this can be computed sequentially in about log n steps by successive squarings. Can it be computed faster than this if we can perform several operations in parallel? KUNG [K1] gave a negative answer to this by showing that even using an unbounded number of parallel processors performing $\{+,\times\ \div\}$ as operations one cannot compute a degree n polynomial in fewer than $\log_2 n$ steps.

For arbitrary univariate polynomials the analysis of parallelism is also tractable. MUNRO and PATERSON [MP] give some precise bounds.

The question becomes more problematic when we have several variables. A special case that can still be computed in O(log n) time is that of arithmetic expressions with $\{+,\times,\div\}$. The essential restrictedness of this model is that the computation graphs are trees rather than arbitrary directed acyclic graphs. BRENT [Br] showed that any such formula of size n can be "rebalanced" so as to have depth O(log n).

For many significant problems, however, no small expression is known. An important example is the determinant of an $n\times n$ matrix $\{x_{11},\ldots,x_{nn}\}$. This has degree n but the "obvious" expression for it has size exponential in n. Classical methods for evaluating determinants rely on elimination techniques, which appear to require n successive stages. In the early 1970's it was quite widely conjectured that this basic process of linear algebra could not be computed in O(n) parallel time. It therefore came as a pleasant surprise when CSANKY [Cs] showed that, at least in fields of characteristic zero, the determinant, and several related problems, could all be computed in $O((\log n)^2)$ parallel steps. Furthermore only polynomially many processors are required [Cs,PrSa].

A second surprise was provided by HYAFIL [Hy]. He showed that any polynomial of degree d that can be computed in C sequential $\{+,\times\}$ steps can be computed by some other program in about (log C)(log d) parallel steps. Thus

any polynomial in n variables that has both degree and sequential complexity polynomial in n can be computed in parallel in time $O((\log n)^2)$. This says that Csanky's fast parallel algorithm is an instance of a more general phenomenon. To see this we note that Gaussian elimination gives a polynomial time algorithm for the determinant, which has degree n, and that divisions can be removed from this algorithm without increasing the complexity more than polynomially [Str,BGH]. Hence Hyafil's result can be applied to this modified algorithm.

The main drawback of Hyafil's construction is that it requires more than polynomial, in fact $n^{\log n}$, parallel processors. A recent result of Valiant, Skyum, Berkowitz and Rackoff [VSBR] remedies this. It shows how any computation using C $\{+,x\}$ operations for computing a polynomial of degree d can be restructured so as to have depth about $(\log C)(\log d)$ and size bounded polynomially in C and d. This seems an encouraging positive result about the potentials for parallel computation. It says that the naive bounds of log d and log C on parallel time can be achieved, to within a product of each other, using only polynomially many processors.

1.2. Boolean Functions.

For discrete computational problems, such as the clique problem discussed in the introduction, the question of finding the right computational model is more problematic than for polynomials. Perhaps the most primitive choice is that of Boolean circuits [Sa]. Given a Boolean function $f(x_1,\ldots,x_n)$ of n arguments, how fast can we compute this in parallel? It is well known that any such function can be expressed as a disjunctive normal form formula of size about 2^n and depth O(n). Although this is a positive statement about depth (i.e. parallel time) it is not useful since it necessitates an exponential number of parallel processors. This observation does explain, however, why the following turn out to be the important questions for parallel Boolean computation:

(i) Which Boolean functions can be computed in o(n) (e.g.,log n) depth?

(ii) Which functions can be computed by circuits that simultaneously have polynomial size and small depth?

The few results that are known can be easily summarized. Boolean expressions of length n can be computed in $o(\log n)$ depth by equivalent expressions. A more interesting class is formed by finite state transducers that produce an output for each input read. Integer addition is an example that can be so formulated. LADNER and FISCHER [LF] show that for any such

transducer the n outputs can be computed from the n inputs by a circuit of size $O(n)$ and depth $O(\log n)$.

A wider class of functions that can be recognized simultaneously in polynomial size and $O((\log n)^2)$ depth was described by RUZZO [R] in terms of alternating Turing machines and contains context-free recognition as a hardest member. It turns out that the class can be characterized in terms of circuit complexity by defining circuits of "polynomially bounded degree" [SV,VSBR]. The restriction here is that implicit conjunction over more than polynomially many terms is never taken.

We have seen already that the degree of a polynomial characterizes fairly accurately its parallel complexity. While no corresponding result is established for Boolean functions, most of the natural problems I know that can be computed simultaneously in polynomial program size and $O(\log n)^k)$ depth can be formulated as instances of the degree result in [VSBR]. For some, such as transitive closure and context-free recognition one first constructs a Boolean circuit of polynomial degree. For others, such as matchings [IMR,SV,BHR,R2] one constructs polynomials, such as the determinant into which the problems can be embedded.

In conclusion, we note that if we impose no restrictions on the class of circuits then reducing depth even slightly appears difficult. Suppose we have a circuit with n gates and say $\Omega(n)$ inputs. At first sight it appears plausible that we can always find a shallow circuit, say depth $\sqrt{n}$ or $\log n$. The only result known [PaVa] states merely that a circuit of depth $n/\log n$ is ensured, and that this is no larger than $2^{n/\log n}$.

1.3 Other Discrete Models

In the theory of sequential computation it has proved useful to study a large number of models of computation and to compare their power. Without this it is difficult to distinguish between theorems that merely describe a specialized property of some particular model, and those that transcend all models. COOK [C4] surveys current knowledge on parallel models.

The question most relevant to the present topic is whether we can parallelize sequential computations more easily if we consider models more powerful than circuits. The obvious directions in which we can enhance the power of a model is to have modifiable connections and to have more complicated operations as a single step.

Examples are the various versions of parallel random access machines [PrSt,FW,LPV,C4]. DYMOND [D] and REIF [R2] show that more parallelization

is indeed possible on such machines. For example a time T computation on a sequential log-cost RAM can be sped up to about $\sqrt{T}.\log(T)$ on a log-cost parallel RAM if $2^{\sqrt{T}}$ processors are available. This result does not translate back to circuits because to simulate one step of parallel random access on $2^{\sqrt{T}}$ processors, depth $\sqrt{T}$ is required. This cancels out the gained saving.

A second question relevant here is whether we can characterize the class of parallelizable functions as a complexity class for some other model of computation. For example, BORODIN [Bo] shows that circuit depth is closely related to sequential Turing machine space. Such relationships have been pursued for many models [PrSt,FW,G,H]. It has been established, for example, that parallel time complexity is indeed a robust notion that is invariant, to within certain factors, for a large class of models. When we try to characterize complexity classes by more than one resource bound simultaneously, such as "polynomial circuit size and O(log n) depth" then the invariances appear to be much weaker [C3,P,Ho].

An interesting general positive result that holds for parallel RAMs but apparently not for circuits is given by REIF [R2]. He shows that a certain Turing machine complexity class can be simulated in O(log n) randomized parallel time. This implies for several graph problems parallel algorithms faster than previously known.

2. GENERAL PURPOSE PARALLEL COMPUTERS

In the previous section we considered in the abstract how algorithms could be devised in which the operations could be performed with much concurrency. To exploit this concurrency in practice we have to implement the algorithm on some specific computer. The question we ask is whether there exist practical architectures on which every parallel algorithms in the above sense can be run with acceptable efficiency. Whether a meaningful answer can be given without reference to technological factors remains to be seen. We find it encouraging, however, that some of the ideas appear to be successful in a surprising variety of settings.

We shall conceptualize a realistic computer as a large number N of universal sequential processors each with some memory, connected by a network of communication lines. Each processor is connected to a small number d of other processors, where d is regarded either as a constant independent of N or a slowly growing function of N such as $\log_2 N$. The processors communicate by sending packets of information along the lines. For simplicity we shall

assume that the transmission time along each wire is the same and dominates the atomic computation steps of the processors. In reality it may, of course, be difficult to engineer this. If transmission times are proportional to physical distance then this requirement can be achieved only for such networks as regular grids. Despite this and other obvious drawbacks the model appears a useful one with which to start and has been discussed widely in the literature.

2.1. Connection Patterns of Small Diameter

The distance d(i,j) from node i to node j in a graph is the minimum number of edges that have to be traversed in any path from node i to node j. The diameter of a graph is the maximum of d(i,j) taken over all choices of pairs i,j. It is clearly advantageous to have as interconnection patterns for the processors graphs of small diameter.

For undirectional transmission (i.e., directed graphs) an optimal solution is easy. Suppose that there are $d \geq 2$ lines directed away from every node and that the number of nodes N is a power of d. Then there are at most $\Sigma_{i=0}^{s} d^i = (d^{s+1}-1)/(d-1)$ nodes within distance s of any one node and hence if s is the diameter then N does not exceed this quantity. It follows that the diameter is at least $\log_d N$. In the positive direction, this bound can be achieved by the directed degree d version of the Bruijn graph (called the d-way shuffle for short). This graph is defined for $N = d^n$ as G = (V,E) where V is the set of vertices $\{(x_1,\ldots,x_n) \mid 1 \leq x_1,\ldots,x_n \leq d\}$ and E the set of edges

$$\{([x_1,\ldots,x_n], [x_0,\ldots,x_{n-1}]) \mid 1 \leq x_0,x_1,\ldots,x_n \leq d\}.$$

For bidirectional transmission (i.e. undirected graphs) the problem is much more difficult. There is no infinite set of values of N for which an optimal solution is known in the above sense. The "Moore bound" has motivated much work [E,HS]. Recent results that attempt to approach this bound as closely as possible are surveyed in [BB,LFQSU].

In the literature much attention has been paid to networks which, though not of optimal diameter, can implement efficiently important algorithms such as the Fast Fourier Transform. Significant examples are the shuffle-exchange [Sto,Sc], the binary n-cube [SBK,VB], and the cube-connected cycles [VuPr] and variations on it [U]. In all cases the diameter differs from the optimal by a constant factor greater than one.

2.2. Parallel Connection Requests

When the information traffic in a network is sparse and packets collide rarely then the minimization of graph diameter is sufficient for efficient communication. In the context of highly parallel computers we expect the information traffic to be very heavy. Suppose we wish to realize a pattern of communication requests in parallel in about $\log_d N$ time units. What patterns of traffic can we expect to cope with?

Clearly if all N nodes send a packet to a single node i simultaneously and if there are at most d lines coming into node i then at least N/d time will be required for all the packets to arrive. We therefore have to exclude such requests. The following four cases exemplify some important patterns that we may hope to achieve in $O(\log_d N)$ time.

(a) permutations: initially one packet at each node, each with a distinct destination address to which it is to be routed.

(b) partial permutations: initially one or zero packets at each node, each with a distinct destination address.

(c) broadcasts: initially one or no packet at each node, each packet with a set of destination addresses to which it or copies of it are to be routed. The sets are disjoint so that at most one packet arrives at any node.

(d) inverse broadcast: these apply when several packets can coalesce into a single packet. Initially there is one or no packet at each node each with a destination address. These addresses do not have to be distinct.

For example if several processors wish to read simultaneously from the memory at a particular node then a broadcast of type (c) would be required. Conversely if several processors wish to write concurrently into a single location with some protocol such as "an arbitrary one of the writes succeeds", then an inverse broadcast (d) suffices. The simplest communication request, the permutation, serves as a useful paradigm for the whole class of these problems.

2.3. Distributed Routing Algorithms

Networks for realizing communication requests have been investigated for many years in the context of telephone switching [Be]. The main characteristic of that application is that telephone conversations are long

i.e. between successive requests to change the communication pattern many bits of information are transmitted. Hence the switching time is of relatively little importance. Permutation networks offer good solutions to many of the problems. A good survey of results for this problem can be found in [MGN].

In multiprocessor interconnection networks we have the additional problem that the communication requests may change at successive time units. Under these conditions the cost of changing the switch setting in standard permutation networks appears prohibitive [GP,LPV,NS,Sc] and other solutions have to be found.

In the search for good routing algorithms it is clearly advisable to keep to those that are fully *distributed*. By this we mean that the algorithm does not rely on any global information about the communication request, and, in particular, the routing of each packet depends only on information available at the node at which it is currently located.

The best distributed routing algorithms known are based on one of two ideas: (i) comparator network sorters [Ba] and (ii) 2-phase randomized routers [V2]. The main advantages of the former are that the local switching computations are simple, being just comparisons of addresses, and that it is deterministic. In contrast the latter require the maintenance of queues of packets at the nodes and use randomization. On the other hand the 2-phase routing algorithms are fast, requiring $c\log_2 N$ basic transmission steps for small c. (N.B. The algorithm is always correct and terminates within this time with large probability for all inputs.) This has been proved analytically in [V2,VB] for the case of the binary n-cube and in [A] and [U] for constant degree graphs, such as the d-way shuffle. Experimental results show that the performance is even better than the anlysis suggests [V4,VB]. In contrast Batchers's construction takes time $\Omega((\log N)^2)$. [Recent work of Ajtai, Komlos and Szemeredi suggests that $C\log_2 N$ time may be possible for very large C for certain degree log N graphs.] A further advantage of the 2-phase router is that, being a natural heuristic, rather than a finely tuned instrument like the Batcher network, it can be used to perform all four of the previously mentioned routing requests without any adaptation. Comparator networks for sorting can directly perform only permutations and need considerable amplifications in order to perform the other tasks.

Recently BORODIN and HOPCROFT [BH] have shown that it is possible to prove some illuminating lower bound results about routing algorithms. They define a routing algorithm to be *oblivious* if the path taken by each packet

depends only on its source and destination. They show that any oblivious deterministic routing algorithm for realizing permutations takes time at least $N^{1/2}/d^{3/2}$. Hence to achieve O(log N) time an algorithm has to be either nonoblivious, like comparator networks, or randomized, like the 2-phase router. A second lower bound result [V3] sheds some light on the 2-phase routing strategy itself. The strategy consists essentially of sending all the packets to randomly chosen nodes in the first phase, and then on to their correct destination in the second. The lower bound gives some justification to this counterintuitive process. The result says that for graphs with near minimal diameter any oblivious randomized algorithm that does not send packets along routes at least twice the diameter of the graph takes more than $\Omega((\log N)^k)$ time for any k.

Finally we note that if we consider regular 2-dimensional grids of diameter $\Omega(\sqrt{N})$ rather than the O(log N) diameter graphs discussed above, then the best way known for realizing permutations are still based on the same ideas. THOMSON and KUNG [TK] adapt Batcher's network so as to work in time $O(\sqrt{N})$ on such a grid. With randomized routing $O(\sqrt{N})$ time with smaller constant multipliers can be obtained [VB]. This underlying unity is encouraging considering that the two kinds of networks are usually associated with very different technological parameters.

REFERENCES

[A] ALELIUNAS, R., *Randomized parallel communication*, Proc. of ACM Symp. on Principles of Distributed Computing, Ottawa, Canada (1982).

[Ba] BATCHER, K., *Sorting networks and their applications*, AFIPS Spring Joint Comp. Conf. 32 (1968) pp. 307-314.

[Be] BENES, V.E., *Mathematical theory of connecting networks and telephone traffic*. Academic Press, New York (1965).

[BB] BERMOND, J.C. & B. BOLLOBAS, *The diameter of graphs - a survey*, Report No. 98, Universite de Paris-sud, LRI, June 1981.

[Bo] BORODIN, A., *On relating time and space to size and depth*. SIAM J. on Computing 6 (1977), pp. 733-744.

[BGH] BORODIN, A., J. VON ZUR GATHEN & J.E. HOPCROFT, *Fast parallel matrix and gcd computations*, Techn. Rep. TR 82-487, Dept. of Computer Science, Cornell University, 1982.

[BH] BORODIN, A. & J.E. HOPCROFT, *Routing, merging and sorting on parallel models of computation*, Proc. of Fourteenth Symp. on Theory of Computing, (1982).

[BM] BORODIN, A. & I. MUNRO, *The computational complexity of algebraic and numeric problems*, American Elsevier, New York (1975).

[Br] BRENT, R.P., *The parallel evaluation of general arithmetic expressions*. JACM 21 (1974), pp. 201-206.

[C1] COOK, S.A., *The complexity of theorem proving procedures*, Proc. Third ACM Symp. on Theory of Computing (1971), pp. 151-158.

[C2] COOK, S.A., *An observation on time-storage tradeoff*, JCSS 9 (1974), pp. 308-316.

[C3] COOK, S.A., *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*, Proc. Eleventh ACM Symp. on Theory of Computing (1979), pp. 338-345.

[C4] COOK, S.A., *Towards a complexity theory of synchronous parallel computation*, L'Enseignements mathematique, T. XXVII, fasc. 1-2 (1981), pp. 99-124.

[Cs] CSANKY, L., *Fast parallel matrix inversion*, SIAM J. on Computing 5 (1976), pp. 618-623.

[DLR] DOBKIN, D. R.J. LIPTON & S. REISS, *Linear programming is log-space hard for P*. IPL 8 :2 (1979), pp. 96-97.

[D] DYMOND, P.W., *Speedup of multi-tape Turing machines by synchronous parallel machines*. Manuscript (1981).

[E] ELSPAS, B., *Topological constrainst on interconnection limited logic*, Switching Circuit Theory and Logical Design 5 (1964), pp. 133-147.

[FW] FORTUNE, S. & J. WYLLIE, *Parallelism in random access machines*, Proc. Tenth ACM Sympt. on Theory of Computing (1978), pp. 114-118.

[GP] GALIL, Z & W.J. PAUL, *An efficient general purpose parallel computer*, Proc. of Thirteenth ACM Symp. on Theory of Computing (1981), pp. 247-256.

[G] GOLDSCHLAGER, L.A., *A unified approach to models of synchronous parallel machines*, Proc. Tenth ACM Symp. on Theory of Computing (1978), pp. 89-94.

[HS] HOFFMAN, A.J. & R.R. SINGLETON, *On Moore graphs with diameters 2 and 3*, IBM J. Res. Dev. 4 (1960), pp. 497-504.

[Ho] HONG, J.W., *On similarity and duality of computation*, Proc. of 21st IEEE Symp. on Foundations of Computer Science (1980) pp. 348-359.

[Hy] HYAFIL, L., *On the parallel evaluation of multivariate polynomials*, SIAM J. on Computing 8 (1979), pp. 120-123.

[IMR] IBARRA, O., S. MORAN & L.E. ROSIER, *A note on the parallel complexity of computing the rank of order* n *matrices*, IPL 11 (1980), p. 162.

[Kuck] KUCK, D.J., et. al., *Measurements of parallelism in ordinary Fortran programs*. Computer 7 (1974), pp. 37-46.

[K] KUNG, H.T., *New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences*, JACM 23:2 (1976), pp. 252-261.

[LF] LADNER, R.E. & M.J. FISCHER, *Parallel prefix computation*, JACM 27 (1980), pp. 831-838.

[LFQSU] LELAND, W.E., R.A. FINKEL, L. QUAO, M.H. SOLOMON & L. ÜHR, *High density graphs for processor interconnection*. IPL 12 (1981), pp. 117-120.

[LPV] LEV, G., N. PIPPENGER & L.G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. on Computers, C-30:2 (1981), pp. 93-100.

[MNG] MASSON, G.M. G.C. GINHER & S. NAKAMURA, *A sampler of circuit switching networks*. Computer, June 1979, 32-48.

[MP] MUNRO, I. & M.S. PATERSON, *Optimal algorithms for parallel polynomial evaluation*. JCSS 7 (1973), pp. 189-198.

[NS] NASSIMI, D. & S. SAHNI, *Parallel algorithms to set up the Benes permutation networks*, IEEE Trans. on Computers, C-31:2 (1982), pp. 148-154.

[PaVa] PATERSON, M.S. & L.G. VALIANT, *Circuit size is non-linear in depth*, TCS 2 (1976), pp. 397-400.

[P] PIPPENGER, N.J., *On simultaneous resource bounds*, Proc. 20th IEEE Symp. on Foundations of Computer Science (1979), pp. 307-311.

[PSt] PRATT, V.R. & L.J. STOCKMEYER, *A characterization of the power of vector machines*. JCSS 12 (1976), pp. 198-221.

[PrSa] PREPARATA, F.P., & D.V. SARWATE, *An improved parallel processor bound in fast matrix inversion*. IPL 7:3 (1978), pp. 148-150.

[PrVu] PREPARATA, F.P. & J. VUILLEMIN, *The cube connected cycles*, CACM 24 (1981), pp. 300-310.

[R1] REIF, J.H., *Symmetric complementation*, Proc. Fourteenth ACM Symp. on Theory of Computing (1982).

[R2] REIF, J.H., *On the power of probabilistic choice in synchronous parallel computations*, Proc. Ninth ICALP, Aarhus, Denmark (1982).

[R] RUZZO, W.L., *On uniform circuit complexity*, JCSS 22 (1981), pp. 365-383.

[Sa] SAVAGE, J.E., *The complexity of computing*, Wiley, New York (1976).

[Sc] SCHWARTZ, J.T., *Ultracomputers*, ACM TOPLAS 2 (1980), pp. 484-521.

[Si] SIEGEL, H.J., *Interconnection networks for* SIMD *machines*. Computer, June 1979, pp. 57-65.

[SV] SKYUM, S. & L.G. VALIANT, *A complexity theory based on Boolean algebra*, Proc. 22nd IEEE Symp. on Foundations of Computer Science (1981). pp. 244-254.

[Sto] STONE, H., *Parallel processing with the perfect shuffle*, IEEE Trans. on Computers, C-20:2 (1971), pp. 263-271.

[Str] STRASSEN, V., *Vermeidung von Divisionen*, J. Reine und Angewandte Mathematik 264 (1973), pp. 184-202.

[SBK] SULLIVAN, H. T.R. BASHKOW, & D. KLAPPHOLZ, *A large scale homogeneous fully distributed parallel machine*, Proc. Fourth ACM Symp. on Computer Architecture (1977), pp. 118-127.

[TK] THOMSON, C.D. & H.T. KUNG, *Sorting on a mesh connected parallel computer*, CACM 20:4 (1977), pp. 263-271.

[U] UPFAL, E., *Efficient schemes for parallel communication*, Proc. ACM Symp. on Principles of Distributed Computing, Ottawa, Canada (1982).

[V1] VALIANT, L.G., *Reducibility by algebraic projections*, Monographie No. 30 de l'Enseignement Mathematique (1982), pp. 365-380.

[V2] VALIANT, L.G., *A scheme for fast parallel communication*, SIAM J. on Computing (1982), to appear. (Also Edinburgh University Comp. Sci. Report CSR-72-80 (1980).

[V3] VALIANT, L.G., *Optimality of a two-phase strategy for routing in interconnection networks*, (1982), to appear.

[V4] VALIANT, L.G., *Experiments with a parallel communication scheme*, Proc. Eighteenth Allerton Conf. on Communication, Control, and Computing, University of Illinois, (1980), pp. 802-811.

[VB] VALIANT, L.G. & G.J. BREBNER, *Universal schemes for parallel communication*, Thirteenth ACM Symp. on Theory of Computing (1981), pp. 263-277.

[VSBR] VALIANT, L.G., S. SKYUM, S. BERKOWITZ & C. RACKOFF, *Fast parallel computation of polynomials using few processors*. To appear. (Preliminary version in Springer Lecture Notes in Computer Science, Vol. 118, (1981), pp. 132-139.)

DESIGN AND COMPLEXITY OF VLSI ALGORITHMS

G.M. Baudet
INRIA, Rocquencourt, France

1. INTRODUCTION

There is no need to convince anyone that Very Large Scale Integration (VLSI) is revolutionizing circuit design. Over the past decade, the level of integration has been regularly doubling every two years or so, and there is no reason to believe that the current trend will stop soon.

Whereas circuit designers ten years ago were faced with the problem of implementing (simple) functions with only a few hundreds of gates, complete micro-processors are now available as commercial products with several hundred thousand transistors on a chip. In another 4 or 5 years, one can expect to have millions of transistors on a chip.

Consequently, conventional measures are no longer adequate for evaluating, and comparing, various circuit designs implementing a given function. In particular, it is commonly admitted that a traditional component count no longer constitutes a valid criterion for measuring a VLSI circuit. A simple reason is that this criterion only accounts for the processing elements of the circuit and does not account for the data transmission between processing elements. Therefore, we cannot expect to capture the full complexity of VLSI circuits through this cost measure.

In the past few years, a systematic approach was taken by MEAD and CONWAY [1980], THOMPSON [1979], and BRENT and KUNG [1980a] to develop new computational models for VLSI circuits. These models were later refined by VUILLEMIN [1980] and LIPTON and SEDGEWICK [1981] and also questioned by CHAZELLE and MONIER [1981]. In these notes, we investigate these different models, and we present some general techniques to establish lower bounds on the complexity of VLSI circuits. Matching upper bounds are illustrated through specific circuit desigms.

This review is organized as follows. In Section 2, we develop a computational model for VLSI and discuss the models that have been proposed in

the literature. In Section 3, we derive direct consequences from the model, and we establish an initial Lemma of THOMPSON [1979, 1980]. This initial result will be used in Section 4 to establish a general lower bound on the class of *transitive functions* (such as binary multiplication, shift, convolution, etc.) [VUILLEMIN 1981]. In Section 5, we use a different technique to develop another lower bound on functions like binary addition, prefix computation, etc. Circuit designs are presented in Sections 6 and 7 to show that the lower bounds can actually be achieved; they also illustrate a very important design methodology based on the use of divide and conquer and of recursive design.

2. A COMPUTATIONAL MODEL FOR VLSI

MEAD and CONWAY [1980], THOMPSON [1979], and BRENT and KUNG [1980a] have laid down the basis for the formalization of a VLSI circuit. Their models differ slightly, and we discuss below their general assumptions.

The basic idea is that a VLSI circuit can be viewed as the layout of a graph in which nodes correspond to gates, and edges correspond to wires. While edges are only used to simulate transmission of signals between nodes, nodes simulate the active part of the circuit and are used to model either the actual processing elements or the I/O ports.

The general assumptions in our model are discussed in the subsequent sections and are organized according to their relevance to logical, technological, electrical, and design assumptions.

2.1. Logical assumptions

Probably the most important (albeit often implicit) assumption in all models is that all signals are supposed to correspond to the digital encoding of pieces of information. Indeed, we would like to make it clear, from the beginning, that the model we are proposing here is certainly not adequate to represent a VLSI circuit as an anolog device; and, as a matter a fact, most of the results developed in these notes would no longer be valid. This is stated in the following.

(L1) A VLSI circuit is a digital device.

The question now is to define the right unit of information that should be used in the model. The only restriction imposed by THOMPSON in his original

paper [1979] was that the values manipulated by the circuit be taken from a finite field, namely the set of integers modulo m. Unfortunately, the modulus m was related to the size N of the problem through $m > N$, causing the lower bound that he derived for the DFT (Discrete Fourier Transform) to be too short by a factor of $\log^{1} N$ (log N being the number of bits required to code integers modulo m).

In agreement with assumtion (L1), signals transmitted along wires (or edges of the graph) will be assumed throughout to represent a boolean value, i.e., exactly one bit of information. Similarly, each node of the graph corresponding to processing elements in our model will be regarded as a boolean function performing some operation on signals adjacent to that node. This is summarized in the following statement.

(L2) A node or a wire can store at most one bit of information.

It is to be noted that the wire transmitting a signal produced by a node carries the same information as the one stored in the node. It is therefore possible to assimilate the wire to the node (but only from the point of view of the quantity of information).

Assumptions (L1) and (L2) will be presupposed throughout the notes, even if they are not explicitly referred to.

2.2. Technological constraints

For a theoretical model to be of any practical value, assumptions should stay close to the real world, and restrictions should also correspond to realistic approximations. We list and discuss below assumptions of the model that are based on current technological constraints. Explicit references to these assumptions will be made in the remainder of the paper. (Notations are drawn mostly from [BRENT and KUNG 1980a, 1981].)

(T1) At most $\nu \geq 2$ wires can overlap at any point in the layout of a VLSI circuit.

This is to account for the fact that all technologies allow for the superposition (without interference) of several wire layers. Typically, ν is a small constant (2 or 3), but it is perfectly conceivable that, with progress in technology, an arbitrarily large number of layers could be superposed, allowing for 3-dimensional chips. We will see in Section 3 how some of the

results can be adapted to VLSI circuits laid out in the space rather than in the plane.

(T2) In one layer, the minimal distance between non-intersecting wires is $\lambda > 0$.

This minimal distance is highly dependent on the technological process used in the fabrication of the chip and reflects the minimal resolution of this process.

(T3) The minimal storage area for one bit of information is $\beta \geq \lambda^2$.

Typically β corresponds to just a few transistors and is in the order of several λ^2.

(T4) The minimal area required by an I/O port is $\rho \geq \beta$.

Although there are no conceptual differences between a gate that corresponds to an I/O port and other logic gates of processing elements, it is convenient to differentiate between them in order to measure their relative importance or influence in the design of a circuit. In addition, if a design corresponds to a full circuit, it has to communicate with the outside world, and I/O ports are typically several orders of magnitude larger than logic gates.

(T5) A bit requires a minimal delay $\tau > 0$ to be transmitted through any gate or I/O port.

With this minimal delay we do not yet consider the propagation delay of a signal along a wire. The assumption concerning this propagation delay is probably the most controversial among the various existing models, and it will be examined and discussed in isolation in the next section.

(T6) Any node has a maximum fan-in $b \geq 2$.

This last assumption is not an absolute requirement and will not be needed directly. It is only a consequence of (T6) together with (T5), derived in Section 3, that will be used to obtain the lower bound of Section 5.

2.3. Propagation delay

While there is general agreement on the assumptions that have been presented in the previous sections, there is little consensus on the correct assumption to put on the time required to propagate a signal along a wire. There are three prevalent assumptions, as discussed in [BILARDI et al. 1981].

The different assumptions are probably best illustrated by considering the time required by a minimal size transistor to charge a wire of length L (or equivalently to propagate a signal along a wire of length L).

(P1) Synchronous model: $t = O(1)$, i.e., the time is independent of L.

(P2) Capacitive model: $t = O(L)$.

(P3) Diffusion model: $t = O(L^2)$.

Assumption (P1) is probably the most commonly adopted in the literature [BRENT and KUNG 1980a, 1981; VUILLEMIN 1980; SAVAGE 1981a], and it is especially useful when proving lower bounds on the computation time of a circuit, since it is the least constrained hypothesis.

Assumption (P3), set forth by CHAZELLE and MONIER [1981], is at least asymptotically, certainly the correct assumption since both the resistance and the capacitance of a wire of length L are proportional to L, and, therefore, the time constant of the wire is quadratic in L. This asymptotic result has, however, no practical application, at least in current technologies, since the quadratic growth would only be sensitive for wires of length several orders of magnitude over the size of feasible chips.

Assumption (P2), on the other hand, is reasonable when considering current electrical parameters: the resistance of a unit width metal wire is quite negligible compared to the resistance of a minimal size transistor. MEAD and REM [1980] have shown that, by appropriately rescaling the transistor, the time delay to charge a wire could be made independent of the size of the wire. In addition, this rescaling corresponds to an area increase of the driving transistor which is directly proportional to the area of the driven wire. Under these conditions, both assumptions (P1) and (P2) can be assimilated.

Assumption (P1) will be used throughout these notes, and the delay introduced by a wire will be included in the delay τ of assumption (T5).

For a full discussion, the reader is referred to the paper of BILARDI et al. [1981].

2.4. Design methodology

The last assumptions that will be considered in our model correspond to geometrical constraints imposed by usual design conventions.

(D1) The layout of a VLSI circuit occupies a convex region of the plane.

Although this assumption is realistic when considering a complete circuit (which always occupies a rectangular area), it might be a restriction for sub-circuits that are to be embedded in larger circuits. LENGAUER and MEHLHORN [1981] have relaxed the convexity assumption and require instead that a chip occupy a *compact* region of the plane. They show that the results that we will develop in Section 4 hold under this new assumption.

The remaining hypotheses are concerned with the input-output schedule.

(D2) Each input is read exactly once.

(D3) Inputs are supplied and ouputs are delivered at fixed times.

(D4) Inputs are supplied and outputs are delivered at fixed locations.

Assumption (D2) states that there is no "free memory" outside the chip, and that, if some input is to be re-used several times, it must be stored within the circuit. In [SAVAGE 1981b, 1982] circuits obeying assumption (D2) are said to be *semelective*.

LIPTON and SEDGEWICK [1981] call circuits satisfying assumption (D3) and (D4) *when-* and *where-oblivious*, respectively, and we will adopt their terminology in these notes. These two assumptions require that the input-output pattern be data-independent (i.e., that it be entirely specified with the circuit design).

Although most circuit designs are where-oblivious (and, therefore, assumption (D4) does not impose any restriction), assumption (D3) excludes all self-timed circuits because outputs are delivered at times that are dependent on the input processed by the chip. It might therefore be useful to distinguish where-oblivious with respect to the input and with respect to the output.

Other authors [LENGAUER and MEHLHORN 1981] have the notion of a

strongly where-oblivious circuit, meaning that the inputs (and outputs) are stored in queues, outside the chip, and only the ordering in the queues are data-independent. Under this assumption, they derive very interesting lower bounds for transitive functions (see Section 4); but we will not be concerned with their results here.

Another restriction which we will not consider in this review, but which has been set forth by several authors (e.g., [CHAZELLE and MONIER 1981]), is that the I/O should take place on the boundaries of the chip. This corresponds to common practice in the fabrication of VLSI circuits but is not required to derive the results reported here.

2.5. Complexity measures

As was already mentioned, a simple component or gate count no longer corresponds to any realistic measure of the size of a circuit when dealing with VLSI circuits. This is particularly true since data communication might, in many cases, represent the largest part of the circuit (in the design of a fast binary shifter, for example, as we will see in Section 4).

In the case of VLSI, the size is better expressed as the total area of silicon used in the layout. The area of a VLSI circuit is throughout denoted by A. Obviously, another crucial parameter of a VLSI circuit is the time T required to compute the function in implements.

The area A and the time T required by a circuit to compute a function constitute important parameters (and have been considered so far the usual measures) of the circuit. These two measures, however, cannot, by themselves, usually capture the full complexity of a VLSI circuit. VUILLEMIN [1981] has introduced a useful complement to these measures with the notion of the *period of a circuit*, which we will denote by P throughout. This period corresponds to the minimal time interval between the input (or the output) of two consecutive instances of a problem solved by the circuit (used in a pipelined fashion).

We feel that the period P of a VLSI circuit is just as important a parameter as its area A and its time T. This is so because the period characterizes the maximum throughput of the circuit, and, in contrast (or in complement) with the time, it is able to take more completely into account the computing power of the circuit. In particular, it measures not the time to solve just one instance of a problem, but the time elapsed between the solutions of two consecutive problems input to a circuit (their execution

taking place simultaneously or in pipeline).

In addition, since any circuit clearly satisfies $T \geq P$, lower bounds on the period P will immediately transfer into lower bounds on the time T, and they will, therefore, correspond to stronger results.

To paraphraze LIPTON and SEDGEWICK [1981], a VLSI circuit designer is typically faced with the following problem (or challenge!).

Given: a boolean function f,

Find: a VLSI layout that computes f and that minimizes all three measures A, P, and T.

Unfortunately, this problem has in general no solution, and, instead of minimizing all three measures together, the circuit designer will have to try to minimize some cost function c(A,P,T) that weights all three fundamental parameters of the circuit.

By assuming that such a cost function is *monotone*, i.e.:

$$c(A',P',T') \geq c(A,P,T)$$

if $A' \geq A$, $P' \geq P$, and $T' \geq T$, and that it is *rescalable* i.e.:

$$c(aA,pP,tT) = g(a,p,t).c(A,P,T),$$

LIPTON and SEDGEWICK [1981] have shown through simple functional analysis arguments that the only cost functions are given by:

$$c(A,P,T) = k.A^a.P^p.T^t.$$

Among such cost functions, some are of particular interest for their physical interpretation. For example, the product A.T measures the energy required by the circuit during the computation of the function it implements, and the product $A.T^2$ measures the corresponding power dissipated by the circuit. Similarly, the product A.P measures the energy required for solving just one instance of a problem (bear in mind that several problems are processed simultaneously by the circuit in a pipelined fashion).

The most common cost functions that have been used in the literature, apart from the direct measures A and T, are the $A.T^2$ measure initially de-

veloped by THOMPSON [1979, 1980] and the $A.P^2$ measure introduced by VUILLEMIN [1980]. Through a combination of lower bound results on both A and $A.T^2$, it is easy to also derive a general lower bound on the product $A.T^{\alpha}$, for $0 \leq \alpha \leq 2$ [THOMPSON 1979, 1980; BRENT and KUNG 1980a, 1981].

3. INITIAL RESULTS

After the description of the VLSI model of computation presented in the previous section, we are now able to derive a few results based on the assumptions of the model.

We will first present a few direct consequences of our notations and assumptions, and we will then re-establish an important lemma initially derived by THOMPSON [1979, 1980] that will serve as the basis for the results of Section 4.

3.1. Direct consequences

LEMMA 3.1. *A circuit with* N *memory cells requires an area at least* $\beta.N$ *and can memorize at most* 2^N *distinct states.*

This first result corresponds to the view of a VLSI circuit as a finite state machine and is a direct consequence of assumption (T3).

LEMMA 3.2. *The time* T *required to compute a function with one output depending on* N *inputs satisfies* $T \geq \tau.\log_b N$.

Again, this is a direct consequence of assumption (T5) and (T6).

It is to be noted that the result of Lemma 3.2 is the only consequence of assumption (T6) that we will require in the remainder of these notes, and the only property that we will actually use is the fact that the computation time T is a *concave* function of N. This property could in fact replace assumption (T6).

In a different model, using assumption (P3), CHAZELLE and MONIER [1981] have obtained a lower bound of $\tau.\sqrt{N}$ for the same computation time T. This relation also corresponds to a concave function and, therefore, could also be used in lieu of $\tau.\log_b N$.

LEMMA 3.3. *Any circuit computing a function of* N *inputs to* M *outputs satisfies:*

$$A.T \geq A.P \geq \rho.\tau.(N+M).$$

Consider a circuit with w_i input ports and w_0 output ports. Then, by (T4) and (T5), $A \geq \rho.(w_i+w_0)$ and $T \geq P \geq \tau.\max(N/w_i, M/w_0)$. □

This is probably the simplest *tradeoff* that can be established for any VLSI circuit, and, although this first lower bound is linear in both the number of inputs and outputs, it appears to be tight in a number of situations (this is the case, for example, of binary addition).

3.2. An initial lemma

Most (if not all) of the early results in VLSI complexity have essentially been derived through a technique initially developed by THOMPSON [1979, 1980]. The principal idea consists of partitioning the circuit into two halves (roughly the same size) and then looking at the necessary flow of information between the two sides of this partition.

The next lemma is a purely geometrical result which is needed in the derivation of Thompson's lemma.

LEMMA 3.4. *Let* C *be the length of any chord perpendicular to a diameter of a convex region of area* A, *then:*

$$A \geq C^2/2.$$

The application of this result to VLSI circuits clearly requires the use of assumption (D1). By assumptions (T1) and (T2), we have the immediate following consequence.

LEMMA 3.5. *Let* ω *be the number of wires crossing any chord perpendicular to a diameter of a convex region of area* A, *then*

$$A \geq \frac{1}{2}.\omega^2.\lambda^2/\nu^2.$$

Let C be the length of the chord. Clearly, from assumptions (T1) and (T2), $C \geq \omega.\lambda/\nu$. The result follows from Lemma 3.4. □

Before stating the last lemma, a few notations and definitions are in order.

Consider a function f, $(z_1, \ldots, z_M) = f(x_1, \ldots, x_N)$, with N inputs and M outputs. Let m be the maximum number of output bits that are delivered through any one output port. Note that, by assumptions (T4) and (T5):

(3.1) $\quad A \geq \rho.M/m$, and $T \geq P \geq \tau.m$.

Consider now a circuit implementing function f and a partition of the circuit by a chord as in Lemma 3.5. Let L and R be the two regions of the circuit resulting from this partition. We will denote this partition by (L,R), and we will also denote by |L| and |R| the number of output variables delivered in each of the two sets L and R, respectively. It is always possible, by appropriately sliding the chord, to arrange that:

$$\lceil (M-m)/2 \rceil \leq |L| \leq |R| \leq \lfloor (M+m)/2 \rfloor .$$

Having defined the partition (L,R) satisfying this last inequality, we denote by f(L1,R1) = (L2,R2) the definition of function f resulting from rearranging the variables according to this partition. We also denote by $f_{L,C}(R1)$ the restriction f(C,R1) of f that consists of setting the variables in L1 to some fixed values C; similarly we define $f_{R,C}(L1)$.

The following definition is adapted from a paper by SAVAGE [1981].

DEFINITION 3.1. Given a function f and a partition (L,R), as above, the *cross-flow* $I_m(f)$ is defined by

$$I_m(f) = \phi_L + \phi_R,$$

where

$$\phi_X = \log \max_C |f_{X,C}|,$$

and where |h| denotes the cardinality of the range of h.

The cross-flow $I_m(f)$ measures the number of bits of information that must be transmitted from one side of the partition (L,R) to the other.

The following lemma is now an immediate consequence of this definition.

LEMMA 3.6 THOMPSON [1979, 1980]. *The area* A, *the time* T, *and the period* P

of any circuit computing some function f *satisfy*

$$A.T^2 \geq A.P^2 \geq k_1.I_m^2(f),$$

where

$$k_1 = (\lambda.\tau/\nu)^2/2.$$

PROOF. As in Lemma 3.5, take a chord that cuts ω wires. Since the computation of f requires the transmission of $I_m(f)$ bits of information across the chord, by assumption (T5), it will take time $T \geq \tau.I_m(f)/\omega$. As for the period P, the chord would constitute a bottleneck as long as $P \leq \tau.I_m(f)/\omega$. The result then follows from Lemma 3.5. □

Since $I_m(f)$ decreases when m increases, this last lower bound is weak for large m. However, by combining the result of this last lemma and equation (3.1), it follows that

$$A.T^2 \geq A.P^2 \geq \max(k_1.I_m^2(f),\ k_2.m.M), \tag{3.2}$$

where

$$k_2 = \rho.\tau^2.$$

ROSENBERG [1981] has considered 3-dimensional circuits in a model similar to the one we have developed in these notes. The results of both Lemma 3.5 and Lemma 3.6 can be transposed immediately in this new 3-dimensional model, and, in particular, the result of Lemma 3.6 becomes

$$V.T^{3/2} \geq V.P^{3/2} \geq k.I_m^{3/2}(f),$$

for some constant k, where V corresponds to the volume of a VLSI circuit and replaces the area A. For more details on 3-dimensional VLSI circuits, the interested reader is referred to the paper by ROSENBERG [1981].

4. A COMBINATORIAL LIMITATION

This section is adapted from a result of VUILLEMIN [1980] which unifies most of the $A.T^2 = O(N^2)$ VLSI complexity results.

Thompson's lemma results in good lower bounds on the $A.T^2$ measure for any function f, provided that the flow of information required by function f is rich enough, i.e., provided that the cross-flow $I_m(f)$ can be shown to be large enough.

VUILLEMIN [1980] considers a class of functions which he called *transitive functions*. Any function in this class enjoys the property that any input bit can be mapped onto any output bit, thus guaranteeing that a maximum of information has to flow inside any circuit that implements such a function.

Let G be a permutation group over {1, ..., N} and let g(1), ..., g(N) be the permutation associated with an element g in G. Formally, he gives the following definition.

DEFINITION 4.1. A function f, $(z_1,\ldots,z_N) = f(x_1,\ldots,x_N,s_1,\ldots,s_p)$, is *transitive of degree* N if there exists a transitive permutation group G over $\{1,\ldots,N\}$ such that, for any g in G, there exist binary values of $s_1,\ldots,s_p$ for which f is permuting $x_1, \ldots, x_N$ according to g, i.e.

$$f(x_1,\ldots,x_N,s_1,\ldots,s_p) = (x_{g(1)},\ldots,x_{g(N)}).$$

The class of transitive functions includes a large number of basic problems. In particular, binary shift, cyclic shift, binary multiplication, convolution, linear transform, etc., can all be shown to correspond to transitive functions. The reader is referred to the paper by VUILLEMIN [1980] for other examples and proofs.

THEOREM 4.1 [VUILLEMIN 1980]. *Any circuit with area* A, *period* P, *and time* T *computing a transitive function of degree* N *satisfies*

$$A.T^2 \geq A.P^2 \geq k.N^2,$$

for some constant k.

PROOF. In this discussion we do not take into account the variables s_i in the definition of function f. Consider a chord partitioning the circuit in

two halves L and R, as in Lemma 3.5. When function f computes a permutation g of the transitive group G, we say that bit x_i crosses the chord if x_i is input in L (resp. R) while $x_{g(i)}$ is output in R (resp. L).

Through simple counting arguments, it can be shown that, during the computation of all permutations in G, the total number of *crossings* must be at least $|G|.\min(|L|,|R|) = |G|.|L|$, where $|G|$ denotes the number of elements in G. Therefore, there must exist a permutation g in G that accounts for at least $|L|$ crossings. This particular permutation contributes to at least $\lceil (N-m)/2 \rceil$ bits of information in the definition of the cross-flow $I_m(f)$.

Theorem 4.1 then follows from equation (3.2). □

5. AN ENTROPIC LIMITATION

Most of the results of this section are exposed in more detail in [BAUDET 1981].

While the technique developed in Section 3.2 and used in Section 4 leads to lower bounds that account exclusively for the amount of wires required by a VLSI circuit, the technique presented in this section leads to lower bounds that account exclusively for the amount of memory required by the circuit. In this respect, these two techniques complement each other.

Throughout the section we regard a VLSI circuit as a finite state machine, and the results are based on the observation of the progress made by a circuit towards the evaluation of the function it implements. At the time of observation, we want to relate the quantity of information still to be produced by the circuit (i.e., the number of bits not yet delivered) to the quantity of information still available from the input (i.e., the number of bits still to be read, together with the information already read and encoded within the circuit).

Before developing these results, we introduce a few notations.

Consider a function f: $f(x_1,\ldots,x_N) = (z_1,\ldots,z_M)$, computed by a circuit C. For the sake of clarity, we assume throughout that M = N. Let $t_i = i.\tau$, for $i \geq 0$, be a sequence of observation times. Let n_i be the number of bits input to the circuit C at time t_i. We define $N_0 = 0$ and $N_i = N_{i-1} + n_{i-1}$, for $i \geq 1$; N_i represents the number of bits input to C prior to time t_i. Also, let s_i be the number of bits encoded within the circuit. Last, let S_i be the set generated by the output variables not yet

delivered by time t_i; the size of this set, denoted by $|s_i|$, corresponds to the number of distinct values that can still be produced by circuit C after time t_i.

If, at time t_i, the evaluation of f is not completed, any output bit not yet deliverd must be evaluated from the $N - N_{i-1}$ input bits not yet read at time t_i, from the n_i input bits just read at time t_i, and from the information memorized within the s_i cells of the circuit (which encodes the input read prior to time t_i). This leads to the following lemma.

LEMMA 5.1. *The area* A *of a circuit computing function* f *satisfies*

$$A \geq \rho.n_i + \beta.s_i,$$

with

$$s_i \geq \log|S_i| - (N-N_i).$$

The first inequality is a direct consequence of assumptions (T3) and (T4). The last inequality follows from the above discussion. □

This lemma already provides us with a lower bound on the area required by a circuit. We cannot expect, however, a lower bound better than linear in the number of outputs produced by function f. Nevertheless, this linear lower bound appears to be tight in a number of cases. This is so, in particular, for any function that is *surjective* and such that any output bit depends on all input bits. This is the case of all transitive functions, and it is stated in the following.

LEMMA 5.2. *The area* A *of any circuit computing a transitive function of degree* N *satisfies:*

$$A \geq N.\min(\rho,\beta).$$

Let t_i be the time when the last input bit is read by the circuit. We have $N_i + n_i = N$. Since any output bit depends on all input bits no output has yet been produced at time t_i, therefore, $S_i = S_0$. The result follows from the fact that the function is surjective (i.e. $|S_0| = 2^N$) and from Lemma 5.1.

Note, that by a combination of Lemma 5.2 and from Theorem 4.1, we obtain a general lower bound on the $A.T^\alpha$ and $A.P^\alpha$ measures. Specifically, we

have

$$A.T^{2x} \geq A.P^{2x} \geq N^{1+x},$$

for $0 \leq x \leq 1$.

The linear lower bound of Lemma 5.2 is based on an instantaneous observation of the circuit. We will show below how to strengthen this result through a continuous sequence of observations over the entire execution of function f.

LEMMA 5.3. *The area* A, *the period* P, *and the time* T *of a circuit computing function* f *satisfy*

$$A.T \geq A.P \geq \rho.\tau.N + \beta.\tau \sum_{0 \leq i\tau < T} \log|S_i| - (N-N_i).$$

The A.T lower bound corresponds to a simple summation of the inequalities of Lemma 5.1 over the entire execution time for one problem (i.e., [0,T]). As for the A.P lower bound, we look at all the problem instances processed simultaneously by the circuit over one period (i.e., [0,P]), and we observe that all the problem instances are independent. □

Again, let us consider a function f which is surjective, i.e., such that $|S_0| = 2^M = 2^N$ (recall that we take M = N). For $i \geq 0$, let M_i be the number of bits that have been produced by the circuit up to (and including) time t_i. Then

$$|S_i| = 2^{N-M_i},$$

and the result of Lemma 5.2 simplifies as

$$A.T. \geq A.P. \geq \rho.\tau.N + \beta.\tau \sum_{0 \leq i\tau < T} (N_i-M_i). \tag{5.1}$$

In particular, equation (5.1) shows that we can obtain a good lower bound on the A.P measure for some function f provided that the quantity (N_i-M_i) can be shown to be large enough. This is the case of functions like binary addition or prefix computation [LADNER and FISHER 1980]. This result is stated in the following.

THEOREM 5.1. *The area* A, *the period* P, *and the time* T *required by any circuit to perform the binary addition of two* N *bit numbers satisfy,*

$$A.T \geq A.P \geq \rho.\tau.N + \frac{1}{2}.\beta.\tau.N.\log_b(N\tau/T).$$

PROOF. Starting with equation (5.1), we want to find an upper bound on the quantities M_i given the sequence N_i. Equivalently, we can find a lower bound on the time t_i at which output bit z_j can be delivered. The proof follows from the fact that, in the case of binary addition, bit z_j depends on all of the input bits x_i for $0 \leq i \leq j$. A detailed proof can be found in [BAUDET 1981]. □

As was mentioned in Section 3.1, we could have derived a similar result in the VLSI model of computaion proposed by CHAZELLE and MONIER [1981]. In their model, the lower bound of Theorem 5.1 becomes

$$A.T \geq A.P \geq \rho.\tau.N + \frac{1}{2}.\beta.\tau.N.\sqrt{(N\tau/T)}.$$

COROLLARY 5.1. *The area* A, *the period* P, *and the time* T *required by any circuit to perform the binary addition of two* N *bit numbers satisfy*

$$A.T^2 \geq A.P.T \geq O(N.\log N).$$

The result follows from Theorem 5.1 and Lemma 3.2, by observing that bit z_{N-1} depends on the N input bits $x_0, \ldots, x_{N-1}$. □

In particular, this re-establishes a result of JOHNSON [1981] on the $A.T^2$ measure.

6. VLSL CIRCUITS FOR CONVOLUTION

In this section, we present a family of VLSI designs for convolution or polynomial multiplication. The results are taken from [BAUDET et al. 1980].

The results not only illustrate some upper bounds that match the lower bounds presented in the previous sections, but also they illustrate a design technique which we think is very important by itself. The designs presented below are based on the use of divide and conquer, and the layouts are built recursively. The same idea was used by LUK [1981] to design a binary multi-

plier.

The convolution of the two number sequences $A = (a_0,\ldots,a_{n-1})$, and $B = (b_0,\ldots,b_{n-1})$ is the sequence $C = (c_0,\ldots,c_{2n-2})$ where

$$c_i = \sum a_j.b_{i-j},$$

for $0 \le i \le 2n - 2$. In other words, convolution is a polynomial product $A(x).B(x) = C(x)$ where $A(x)$, $B(x)$, and $C(x)$ are the polynomials with coefficients (a_i), (b_i), and (c_i), respectively.

We will assume that the input coefficients a_i and b_i are in the range $[-2^{p-1}, 2^{p-1}-1]$. The output coefficients c_i need, therefore, to be coded over $k = 2p + \lceil \log n \rceil$ bits. By padding input coefficients with sufficiently many zeroes, we can assume that both input and output coefficients are coded over k bits. Measured as the total number of output bits, the size of our problem is $N = O(n.k)$. We observe that $k \ge \log N$.

The most widely used circuit for convolution involves a multiplier accumulator. Although it only requires a small area, linear in the problem size (which is the best we can achieve), its computing time is too large and is prohibitive in many applications.

Another solution is presented in [KUNG and LEISERSON 1983], using a *systolic* design. In their design, both the area and the computing time are linear in the problem size.

The key idea in the designs presented below is the recursive construction of the circuits. The simplest form is presented here. Refinements on these designs and optimal circuits are presented in [BAUDET et al. 1980].

The polynomial product, $C(x) = A(x).B(x) = c_0 + c_1x + c_2x^2$, of two degree one polynomials, $A(x) = a_0 + a_1x$ and $B(x) = b_0 + b_1x$, can be computed in 4 multiplications and 1 addition

$$(6.1) \qquad p_1 = a_0.b_0,\ p_2 = a_0.b_1,\ p_3 = a_1.b_0,\ p_4 = a_1.b_1,$$
$$c_0 = p_1,\ c_1 = p_2 + p_3,\ c_2 = p_4.$$

In general, the polynomial product of two degree n polynomials can also be computed with 4 products of polynomials of degree n/2 and with 4 additions of of polynomials of degree n/2, by simply splitting the coefficients into two halves.

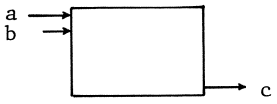


Figure 6.1: A basic serial multiplier.

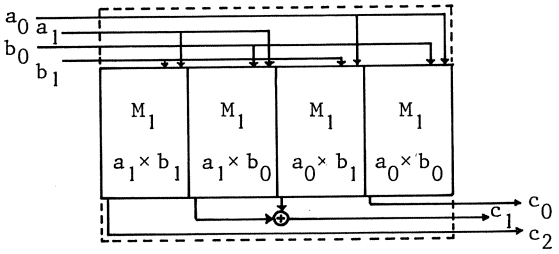


Figure 6.2: Circuit M_2.

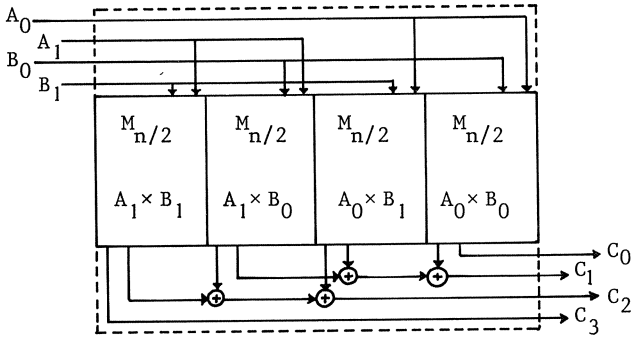


Figure 6.3: A circuit M_n constructed from 4 circuits $M_{n/2}$.

These decompositions lead naturally to the recursive construction of a polynomial multiplier. The basis of the recursion is a circuit M_1 (shown in Figure 6.1) which performs the serial multiplication of two k-bits integers a and b, computing serially the k bits of c = a.b. Then formulas (6.1) can be used to define a circuit M_2 (see Figure 6.2) combining 4 circuits M_1 and 1 elementary serial adder.

In general, a circuit M_n can be constructed using 4 circuits $M_{n/2}$ and 4 serial adders, as shown in Figure 6.3. There each wire represents n/2 parallel wires, each of which carries k bits serially. The serial adders have an area the same order of magnitude as the wires themselves.

The performances of this circuit can be obtained readily from the recursive construction. Let A_n, P_n, and T_n denote the area, the period, and the time of circuit M_n, and let W_n and H_n be its width and its height (so that $A_n = W_n.H_n$). For $n > 1$, we have

$$W_n = 4.H_{n/2},\ H_n = W_{n/2} + 8.\frac{n}{2}.\lambda,$$

$$P_n = \max(P_{n/2}, k.\tau),\ T_n = T_{n/2} + k.\tau,$$

where $k.\tau$ is the time for the serial addition.

The basis for these recurrences is provided by the performances of circuit M_1: $A_1 = W_1.H_1 = a.k$, $P_1 = T_1 = t.k$, where a and t depend on the actual design of M. We obtain:

$$A_n = (4.\lambda.n.\log n + n.W_1).(2.\lambda.n.\log n + n.H_1),$$

$$P_n = \max(t.k,\ \tau.k),$$

$$T_n = \tau.k.\log n + t.k.$$

Expressed as the total problem size $N = n.k$, we deduce that:

$$A.P^2 = O(N^2.k^2.\log^2 N) \quad \text{for} \quad k \leq \log^2 N,$$

$$A.P^2 = O(N^2.k^3) \quad \text{for} \quad k \geq \log^2 N.$$

This circuit is therefore not optimal with respect to the $A.P^2$ measure. Another circuit, based on a different multiplicative scheme, is presented in [BAUDET et al. 1980], and this circuit is optimal with respect to this measure.

7. VLSI CIRCUITS FOR ADDITION

The problem investigated in this section is the binary addition of two n-bit numbers: given $A = a_0 + 2.a_1 + \ldots + 2^{n-1}.a_{n-1}$, and similarly for B, find $S = A + B = s_0 + 2s_1 + \ldots + 2^n s_n$.

There are two classes of binary adders, depending upon the time required to perform the addition of two n-bit integers.

Slow adders perform a binary addition in a time which is linear in the

length of the integers. The basic building block is the *full adder* of Figure 7.1 (see end of section). Its function is to compute the sum of the three input bits: a,b,c_{in} and to provide a 2-bit representation of this sum: s, c_{out}. Specifically

$$2.c_{out} + s = a + b + c_{in},$$

so that

$$s = a + b + c_{in} \pmod 2,$$
$$c_{out} = (a+b+c_{in}) > 1.$$

Such adders are called *carry-chain* or *carry-propagate* adders and are inherently sequential since the evaluation of bit s_i requires the knowledge of the carry in position $i - 1$.

Although all carry-propagate adders perform a binary addition in time $T = O(n)$, it is possible to enhance their performance through pipeline. In particular, the circuit of Figure 7.2, which we could term a systolic adder, is fully pipelined (i.e., it functions at clock rate). Its performances are

$$A = O(n),\ P = O(1),\ \text{and}\ T = O(n).$$

Through a combination of the two circuits of figures 7.1 and 7.2, it is quite simple to design a carry-propagate adder which has performances $A = O(n/p)$, $P = O(p)$, and $T = O(n)$ for any p in $[1,n]$. The parameter p is a measure of the degree of parallelism and corresponds to the number of problems processed simultaleously

This circuit also shows that it is not possible to improve on the linear lower bound of Lemma 3.3.

On the other hand, fast adders allow the binary addition of two integers in sub-linear time (in our model). Such adders are called *carry-save* or *carry-lookahead* adders.

We will report below the design of a fast adder proposed by BRENT and KUNG [1980a].

In terms of boolean equations, the carry and sum in position i can be computed as follows

$$(7.1)\qquad \begin{aligned} c_i &= (a_i \cdot b_i) + (a_i \cdot c_{i-1}) + (b_i \cdot c_{i-1}), \\ s_i &= a_i \otimes b_i \otimes c_{i-1}, \end{aligned}$$

where the operators., +, and ⊗ denote the *and*, *or*, and *exclusive-or* boolean operations respectively. (We define $c_0 = 0$.)

The evaluation of c_i can be rewritten as

$$c_i = g_i + (p_i \cdot c_{i-1}),$$

with

$$g_i = a_i \cdot b_i, \text{ and } p_i = a_i \otimes b_i.$$

Define the operator * by

$$(g,p) * (g',p') = (g + (p \cdot g'),\ p \cdot p'),$$

and let

$$\begin{aligned} (G_0,P_0) &= (g_0,p_0) \\ (G_i,P_i) &= (g_i,p_i) * (G_{i-1},P_{i-1}) \quad \text{if} \quad i \geq 0. \end{aligned}$$

BRENT and KUNG [1980a] have shown that $c_i = G_i$. Since the operator * is associative, all the (G_i,P_i), and therefore all of the carries c_i, $0 \leq i < n$, can be computed in log(n) parallel steps.

The circuit of Figure 7.3 corresponds exactly to this evaluation. All the output bits s_i, $0 \leq i \leq n$, can then be evaluated through equation (7.1).

We observe that this circuit can be fully pipelined, in which case its performances are given by

$$A = O(n.\log n),\ P = O(1),\ T = O(\log n).$$

In particular, we notice that the circuit is optimal with respect to the A.P.T measure while it is not with respect to the $A.T^2$ measure.

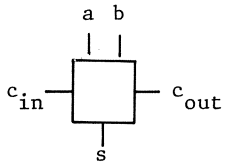


Figure 7.1: A Full-Adder.

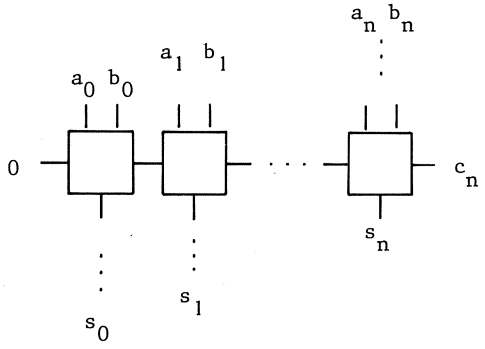


Figure 7.2: A systolic adder.

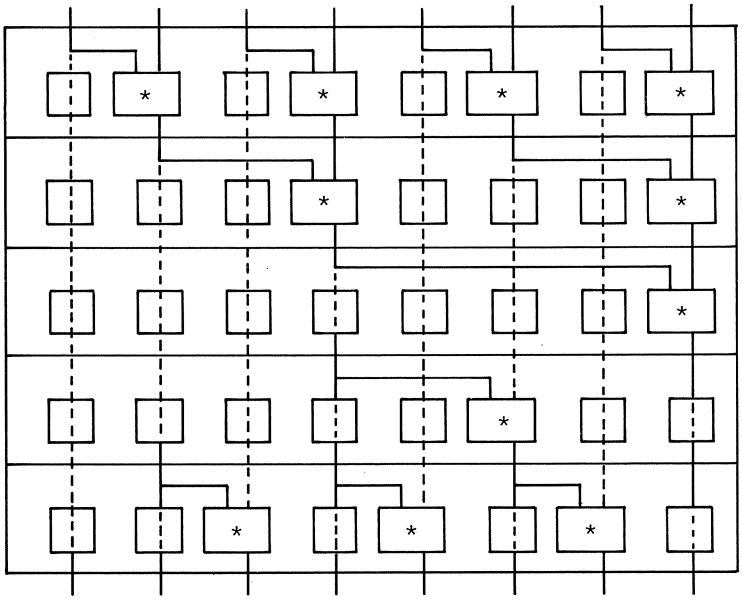


Figure 7.3: A fast adder.

REFERENCES

ABELSON, H., and ANDREAE, P. (1980), *Information transfer and area-time tradeoffs for* VLSI *multiplication*, Communications of the ACM, Vol. 23, No. 1, January 1980, pp. 20-23.

BAUDET, G.M. (1981), *On the area required by* VLSI *circuits*, in VLSI *Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds), Computer Science Press, October 1981, pp. 100-107.

BAUDET, G.M., PREPARATA, F.P., and VUILLEMIN, J.E. (1980), *Area-time optimal* VLSI *circuits for convolution*, Technical Report, No. 30, INRIA, Rocquencourt, August 1980. (To appear in IEEE Transactions on Computers.)

BILARDI, G., PRACCHI, M., and PREPARATA, F.P. (1981), *A critique and an appraisal of* VLSI *models of computation*, in VLSI *Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds), Computer Science Press, October 1981, pp. 81-88.

BRENT, R.P., and KUNG, H.T. (1980a), *The chip complexity of binary arithmetic*, Proceedings of the 12-th Annual ACM Symposium on Theory of Computing, April 1980, pp. 190-200.

BRENT, R.P., and KUNG H.T. (1980b), *On the area of binary tree layouts*, Information Processing Letters, Vol. 11, No. 1, August 1980, pp. 46-48.

BRENT, R.P., and KUNG H.T. (1981), *The area-time complexity of binary multiplication*, Journal of the ACM, Vol. 28, No. 3, July 1981, pp. 521-534.

CHAZELLE, B., and MONIER, L. (1981), *A model of computation for* VLSI *with related complexity results*, Proceedings of the 13-th Annual ACM Symposium on Theory of Computing, May 1981, pp. 318-325.

GUIBAS, L., and VUILLEMIN, J. (1982), *On fast binary addition in* MOS *technologies*, in preparation.

HONG, J.W., and KUNG H.T. (1981), I/O *complexity: The red-blue pebble game*, Proceedings of the 13-th Annual ACM Symposium on Theory of Computing, May 1981, pp. 326-333.

JOHNSON, R.B., Jr. (1980), *The complexity of a* VLSI *adder*, Information Pro-

cessing Letters, Vol. 11, No. 2, October 1980, pp. 92-93.

KEDEM, Z.M., and ZORAT, A. (1981), *On relations between input and communication/computation in* VLSI, Proceedings of the 22-nd Annual IEEE Symposium on Foundations of Computer Science, October 1981, pp. 37-44.

KUNG, H.T. (1979), *Let's design algorithms for* VLSI *Systems*, Proceedings of the Caltech Conference on VLSI, January 1979, pp. 65-90.

KUNG, H.T., and LEISERSON, C.E. (1980), *Systolic arrays (for* VLSI), *in* [MEAD and CONWAY 1980], pp. 271-292.

LADNER, R.A., and FISHER M.J. (1980), *Parallel prefix computation*, Journal of the ACM, Vol. 27, No. 4, October 1980, pp. 831-838.

LEIGHTON, F.T. (1981), *New lower bounds techniques for* VLSI, Proceedings of the 22-nd Annual IEEE Symposium on Foundations of Computer Science, October 1981, pp. 1-12.

LENGAUER, T., and MEHLHORN, K. (1981), *On the complexity of* VLSI *computations*, in VLSI *Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds), Computer Science Press, October 1981, pp. 89-99.

LIPTON, R.J., and SEDGEWICK, R. (1981), *Lower bounds for* VLSI, Proceedings of the 13-th Annual ACM Symposium on Theory of Computing, May 1981, pp. 300-307.

LIPTON, R.J., and TARJAN, R.E. (1980), *Applications of a planar separator theorem*, SIAM Journal on Computing, Vol. 9, No. 3, August 1980, pp. 615-627.

LUK (1981), *A regular layout for parallel multiplier of* O((log n)**2) *time*, in VLSI *Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds), Computer Science Press, October 1981, pp. 317-326.

MEAD, C., and CONWAY, L. (1980), *Introduction to* VLSI *Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

MEAD, C., and REM, M. (1980), *Highly concurrent structures with global communication, in* [MEAD and CONWAY 1980], pp. 313-329.

PREPARATA, F.P., and VUILLEMIN, J.E. (1980a), *Area-time optimal* VLSI *networks*

based on the Cube-Connected-Cycles, Technical Report, No. 13, INRIA, Rocquencourt, March 1980.

PREPARATA, F.P., and VUILLEMIN, J.E. (1980b), *Area-time optimal* VLSI *networks for multiplying matrices*, Information Processing Letters, Vol. 11, No. 2, October 1980, pp. 77-80.

PREPARATA, F.P., and VUILLEMIN, J. (1981), *The Cube-Connected Cycles: A versatile network for parallel computation*, Communications of the ACM, Vol. 24, No. 5, May 1981, pp. 300-309.

ROSENBERG, A.L. (1981), *Three-dimensional integrated circuitry*, in VLSI *Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds), Computer Science Press, October 1981, pp. 69-80.

SAVAGE, J.E. (1981a), *Area-time tradeoffs for matrix multiplication and related problems*, Journal of Computer and System Sciences, Vol. 22, No. 2, April 1981, pp. 230-242.

SAVAGE, J.E. (1981b), *Planar circuit complexity and the performance of* VLSI *algorithms*, in VLSI *Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds), Computer Science Press, October 1981, pp. 61-68.

SAVAGE, J.E. (1982), *Bounds on the performance of multilective* VLSI *algorithms*, Department of Computer Science, Brown University, Technical Report No. CS-82-10, March 1982.

THOMPSON (1979), *Area-time complexity for* VLSI, Proceedings of the 11-th Annual ACM Symposium on Theory of Computing, May 1979, pp. 81-88.

THOMPSON, C.D. (1980), *A Complexity Theory for* VLSI, Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, August 1980.

VUILLEMIN, J. (1980), *A combinatorial limit to the computing power of* VLSI *circuits*, Proceedings of the 21-st Annual IEEE Symposium on Foundations of Computer Science, October 1980, pp. 294-300.

YAO, A.C. (1981), *The entropic limitations on* VLSI *computations*, Proceedings of the 13-th Annual ACM Symposium on Theory of Computing, May 1981, pp. 308-131.

SYSTOLIC COMPUTATION AND VLSI

M.R. Kramer & J. van Leeuwen
University of Utrecht, Utrecht, the Netherlands

ABSTRACT

The notion of systolic computation is due to H.T. Kung and C.E. Leiserson. It describes the computation by means of a network of simple processing elements that rhythmically act on regular streams of data passing through the system. We explain the paradigms of systolic algorithm design through a discussion of systolic queues, stacks and trees. An integral approach is given to matching problems and matrix multiplication performed by (2-dimensional) systolic arrays. As a novel contribution we present a systolic algorithm for inverting a nonsingular $n \times n$ matrix in $O(n)$ time.

1. INTRODUCTION

With the advent of VLSI-technology (cf. Mead and Conway [17]) it has become feasible to design circuits with tens of thousands of components and integrate them on a single chip of silicon. The large degree of parallelism that can be incorporated in circuits of this size gives VLSI-chips at least the potential of providing extremely fast and efficient computing devices. The development of systolic algorithms as initiated by Kung and Leiserson [13] can be viewed as an approach aimed at exploiting this potential through the use of modular design principles and fully parallel and pipelined data processing (streaming).

A VLSI-chip as understood here is described by its constituent components (processing elements) and their interconnections (the wiring pattern). We will simply assume that processing elements consists of a single "cell",

even though they may be built of several transistors and like components. To facilitate the design of the chip, we insist that the actual variety of different processing elements used be kept as small as possible. The wiring pattern needs to be simple and regular for the same reason, with only local connections of processing elements and no long wires that need more area and more energy to drive and that are invariably slower. The algorithm performed on the chip should exploit the parallel processing power of the many available cells, which includes both a "division of labor" and the use of pipelining through the circuit.

The VLSI-chips that conform to these rules are characterized by simple geometries (arrays) of cells, with the cells rhythmically acting on one or more streams of data that smoothly move across the chip. The algorithms underlying the operation of the chip have been termed "systolic" (Kung and Leiserson [13]) for the analogy with the rhythmic pulsing of blood through the arteries. Systolic algorithms and architectures have been studied extensively by H.T. Kung (see e.g. [10], [11]) and several co-workers. Independently similar goals have been pursued by the research group of T. Legendi (see e.g. [14]) in the study of regular "fields" of cellular processors, a hardware oriented outgrowth of the theory of cellular automata as known from e.g. [5].

In these notes we shall illustrate the paradigms of systolic algorithm design through a number of examples that, aside from being of interest in their own right, do provide some useful special circuits for inclusion in more involved designs. In Section 2 we discuss a design for systolic priority queues due to Leiserson [15] and show how it can be used to sort n keys in $O(n)$ time and to implement ordinary queues and stacks with $O(1)$ response times. In Section 3 we discuss the "tree machine" as proposed by Bentley and Kung [3] and the systolic trees of Leiserson [15], aimed again at the implementation of a fast priority queue. We analyse some shortcomings of both designs and the solutions for it proposed by Song [21] and Ottmann, Rosenberg and Stockmeyer [18], respectively. In Section 4 we provide a uniform treatment of pattern matching (as in Foster and Kung [8]), comparison problems (as in Kung and Lehmann [12]) and matrix multiplication (as in Kung and Leiserson [13] and Katona [9]), which all use a similar idea of pipelining on a 1- or 2-dimensional array of cells. In Section 5 a (novel) systolic algorithm is presented to invert an $n \times n$ nonsingular matrix in $O(n)$ time. The algorithm is based on Gaussian elimination and assumes that no pivoting is required. The algorithm was proposed for

implementation on parallel architectures before (Pease [19]), but the systolic version presented here shows the intricacy of pipelining to achieve greater speed. Each of the Sections demonstrates a particular way of describing the actions of a systolic circuit: operational in terms of actions of cells on their contents (Section 2), programmed for a small repertoire of instructions (Section 3), operational in terms of actions on the data streams (Section 4) and functional (Section 5, as in the framework of e.g. Katona [9]).

From an abstract point of view, systolic algorithms are not very different from multi-processor algorithms with the processors acting in fully synchronized order. Two levels of timing can be distinguished in the design of a systolic algorithm: (i) the global timing required to synchronize the cells as a network, and (ii) the local (internal) timing required for the operation within a cell. The latter is not trivial, for we shall design processing elements to perform non-elementary operations (like multiplication) on b-bit numbers. Nevertheless we shall view these operations as atomic and requiring only one "tick" of the global clock.

2. LINEAR SYSTOLIC QUEUES AND STACKS

We shall primarily discuss the design of a *systolic priority queue*, i.e., a structure that supports INSERT/DELETE/XMIN ("extract smallest key") commands on a set that never contains more than N keys. The design consists of a linear array of cells (see figure 1) with its I/O connection to the environment left of the first cell. Every cell has two registers, A and B, each of which can contain a key of the set. If

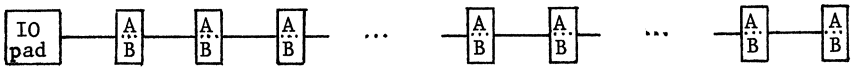


figure 1

a register carries no key, we assume it contains a default value (∞) larger than any conceivable key. Our aim is to maintain the set of $n \leq N$ keys in increasing order, with all keys in the A-registers of a contiguous initial segment of the array. The B-registers are used for transporting newly

inserted keys to their proper position in the ordering. If a key got to its place, it moves the current key out of the A-register and replaces it, while the latter takes over in "streaming" right. It should be clear that an insertion thus has an effect that ripples on through the array, moving all subsequent keys one place up. As soon as one insertion has moved one step one can initiate a next one, for the first cell would now be idle.

A deletion would require a special signal to be sent up the array, to search for and delete a particular key. All keys to the right should subsequently move down one place, which they will appear to do one after the other and just fast enough to give instructions coming in from the left an undistorted impression of contiguity of the data set. The extraction of the smallest key is a special case, with an additional instruction to send the key (always in the A-register of the first cell) left to output. For uniformity we only program the array to do the latter kind of deletion.

The cells will be timed such that the odd and even numbered cells "beat" alternately. When it acts, a cell will compare its register contents with the register contents of its neighbour to the left. The timing of the array is such that this neighbour is momentarily in-active and (thus) it is safe to inspect it. The "IO pad" (see figure 1) should act as a left neighbour when it has to. At other times it can route keys in and out. Let A_ℓ and B_ℓ denote the register contents of the A and B registers of a left neighbour, respectively. The cells will cycle through the following program:

```
do
  cycle 1: copy B_ℓ into B; rearrange the keys in A_ℓ, A and B such
           that A_ℓ ≤ A ≤ B;
  cycle 2: rest
od.
```

Observe that values in a B-register indeed move right and are swapped in place when the proper place in the ordering ("before the current A-value") is reached. If A_ℓ contains ∞ (ultimately the result of a deletion or XMIN), then keys right of it will move a step down (in ripple fashion). The copying of a B_ℓ-register (in cycle 1) could, but need not destroy the value of this register. The IO-pad has an A and a B part that is set as follows, to interact properly with the first cell of the array and allow for the processing of commands:

(i) to insert a key k, A is set to $-\infty$ and B to k (which thus prepares it for moving up in the B-registers),

(ii) to extract a smallest key, both A and B are set to ∞. In the next cycle (or the one after that, depending on the timing of the array) the smallest key will have moved into the A-register and can be extracted. The first cell now has a value ∞ in its A-register, which will be filled from the right immediately in the following step,

(iii) when idling, A is kept at $-\infty$ and B at ∞.

The design has no provision for signaling when the queue is "full". Overflow will result in the loss of elements at the end. The following conclusion should be evident ([15]).

THEOREM. *A linear systolic array of size* N *can process* INSERT/XMIN *(and* DELETE*) commands with* $O(1)$ *response times, as long as the number of keys in the set remains* $\leq$ N *at any moment.*

The design immediately leads to a fast on-line sorter which operates in $O(n)$ time on a set of n keys. Just feed the keys into the queue and extract the smallest from it in n consecutive steps. (The method of Armstrong and Rem [2] is a special case of this for bitwise sorting).

The principle of the systolic priority queue can be used to implement ordinary queues and stacks. A straightforward idea is to stamp elements with a natural number, which gives their ordering in the queue or stack, and to subsequently use it for a key to move the elements into the array. We do not propose that this be used but merely note that it is a convenient way of conceptualizing the required data movements through the A and B registers. The *systolic queue* is like a systolic priority queue in which the elements are kept sorted in inverse order of arrival. Thus, new keys always move up through the B-registers until the first open A-register at the end of the line is reached. Deletions from the front of the queue are executed exactly like the XMIN command. We no longer need ∞ for reasons of ordering and simply use it to mean that a register is "empty". The cells of the systolic queue can abide by the following simple program (recall that the odd and even numbered cells still alternate in action):

```
do
  cycle 1: if
             A_ℓ & B_ℓ & A → copy B_ℓ into B;
           □ A_ℓ & B_ℓ & ¬ A → copy B_ℓ into A; set B to ∞;
           □ ¬ A_ℓ & ¬ B_ℓ & A → copy A into A_ℓ; set A and B to ∞
           fi;
  cycle 2: rest
od.
```

(The if ..□..□.. fi is a guarded command, which effectively acts like a *skip* if no guard happens to be satisfied. The &-notation is a simple shorthand for "is not empty and".) Note that, as deletions occur at the front of the queue, there will never be more than one empty A-register in between two occupied ones.

THEOREM. *A linear systolic array of size* N *can process* ENQUEUE/DEQUEUE *commands with* $O(1)$ *response times, as long as the number of elements in the set remains* $\leq$ N.

A *systolic stack* is like a systolic priority queue in which elements are kept sorted in exact order of arrival. Thus, a newly inserted key always forces the element in the A-register of the first cell to move up, which in turn forces all elements in the queue to step one position to the right. A deletion is again very similar to the XMIN command and causes all elements to do one step left. The cells of a systolic stack, again, can do with a simplified program:

```
do
  cycle 1: if
             A_ℓ & B_ℓ & A → copy A into B; copy B_ℓ into A;
           □ A_ℓ & B_ℓ & ¬ A → copy B_ℓ into A; set B to ∞;
           □ ¬ A_ℓ & ¬ B_ℓ & A → copy A into A_ℓ; set A and B to ∞
           fi;
  cycle 2: rest
od.
```

THEOREM. *A linear systolic array of size* N *can process* PUSH/POP *commands with* $O(1)$ *response times, as long as the number of elements in the set remains* $\leq$ N.

3. SYSTOLIC TREES

With the many uses of trees in search structures (cf. [1]) it seems advantageous to connect N processing elements by a tree-based circuit rather than in a linear array. Again the cells can have several

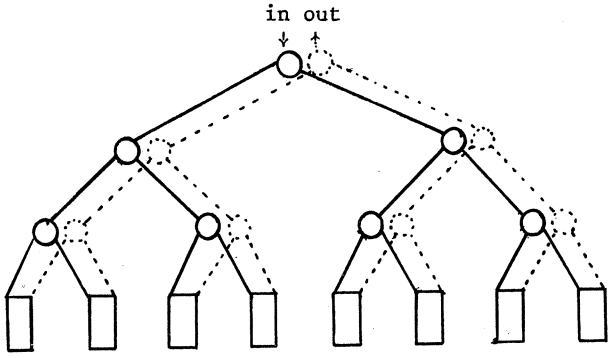


figure 2

registers, but for the moment we assume that each cell can hold at most one key. The tree-circuit (see figure 2) is best thought of as consisting of two strata: one to route information down to the cells, and another to merge information and route it upward. The O-nodes simply split or duplicate messages in one step, the ⬡-nodes are more complicated and can perform some function on the two incoming messages (data) from below to determine what message to pass upward. Packaging a tree machine on a small chip is a non-trivial matter discussed at some lenght in Song [21] and Bhatt and Leiserson [4].

The important characteristic of the tree machine is that it allows for extremely efficient broadcasting of information to all processors simultaneously. Assuming the processors all act in one step, the result information can be merged and sent back up the tree equally fast. (One might call it "inverse broadcasting"). In the applications that follow we shall assume that the machine is pipelined. This means that there is a steady movement of wavefronts down the O-tree and up the ⬡-tree, with every wavefront carrying its own information (corresponding to one command) and spanning an entire level of the tree. A distinction should be made between

the compute time (i.e., the time between submitting a request and receiving the corresponding output) and the period of a command (i.e., the time between its wavefront and the next, or: the interval of time that must pass before a next command can be entered).

METATHEOREM. A tree machine of size N can process any admissible command with a compute time of $O(\log n)$ and a period of $O(1)$, as long as the number of elements in the set remains $\leq N$.

The metatheorem does not always hold, but it serves as a criterion to test the suitability of the tree machine for a particular set of commands. We shall try an implementation of a simple dictionary, i.e., a structure that supports MEMBER/INSERT/DELETE commands on a set that never contains more than N keys. *We assume throughout that keys are unique* and (thus) that there never occur two identical keys in the tree. It means there never is an insertion of a key that is already present. (It only points to the first of several problems inherent to the systolic approach...)

A MEMBER(k) command is easy. Broadcast it to all processors, let them answer yes or no depending on the key they contain and merge the answers up the tree to check that there was at least one "yes". It takes a compute time of $O(\log N)$ and a period of $O(1)$. An INSERT(k) is harder, because there is no point in broadcasting the instruction and store k in every available (free) A-register as a result! To guide the search for a free register, one could add to each O-node a counter that keeps track of the number of free registers in the cells it covers in its subtree. The INSERT command is moved in a direction with count > 0, while the counters that it passes are (of course) immediately decreased by 1, until it eventually reaches one free register where k gets stored. It complicates the tree machine tremendously, but does give an $O(\log N)$ compute time and an $O(1)$ period. To process a DELETE(k) one would need to update the counters along the search path towards k only. A DELETE thus causes severe problems, because the search path can only be traced during the wave back up the tree, i.e., after having located k. This forces the period for DELETE commands to remain at $O(\log N)$, apparently.

THEOREM. *A tree machine of size* N *can process* MEMBER/INSERT/DELETE *commands with a compute time of* $O(\log N)$ *and a period of* $O(1)$, *as long as the number of elements in the set remains* $\leq N$. *(It is assumed that* INSERTs *always add new keys and that* DELETEs *always remove existing keys.)*

PROOF. The argument is due to Song [21]. Suppose the B-register of every processor is made available to hold any integer $\in 1..N$ or ∞. If a processor holds a key, then its B-register contains ∞ ("empty"). If it holds no key, then the B-register contains a value that essentially is its position in a free space list. A (global) counter F is maintained at the root of the machine to keep track of the total number of free A-registers. If F has value f, the "tickets" 1 to f are somehow distributed over (i.e., contained in the B-registers of) the now available processors. An INSERT(k) command is tagged with the current value of F and broadcasted to all processors. The value of F is rightaway decreased by 1 and the actual insertion of k only takes place in the processor whose B-register holds ticket f. (The B-register is subsequently erased). A DELETE(k) command is tagged with the current value of F plus 1 and broadcasted down. The value of F is incremented and the processor that contains k stores the tag in its B-register. (This time the A-register is, of course, erased.) □

The problem of handling extraneous insertions/deletions (called "redundant" insertions/deletions in [18]) remains, at least for the time being.

The counting of free processors became necessary, because we apparently lost the possibility of shifting keys left and right in the array to maintain the set in a contiguous initial segment of the processors. Define an L-machine to be a tree machine of some size N with the processors connected into a linear array (see figure 3).

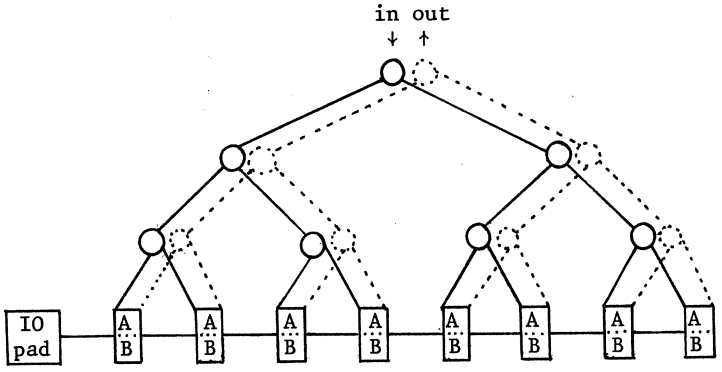


figure 3

We assume as in Section 1 that an IO-pad marks the left end of the array. The L-machine retains the facility of the tree machine to broadcast instructions to all processors but combines it with the attractive feature of linear arrays to move keys among neighboring cells. We shall assume that the wavefronts activate the processors simultaneously and do not succeed one another faster that the processors need to complete their cycle. The packaging of L-machines on one or more chips is a bit harder but can be done along the same lines as for tree machines (cf. Song [21]). We shall exploit the L-machine to implement an extended dictionary structure that supports MEMBER/INSERT/DELETE/XMIN commands. We shall ignore MEMBER commands, as they are treated exactly as for tree machines.

Commands are broadcasted from the root to all processors. Depending on the instruction received the processors shift keys right (if they recognize that an insertion took place to their left) or left (if they recognize that a deletion took place to their left or the command was an XMIN) or not at all (if they recognize there is no need for it). The processors can decide the required action by comparing k to their A-register and the A-register of their left neighbor. When timed right, the machine maintains the set of keys in increasing order in an initial segment of the array. In particular, an XMIN will automatically shift the smallest key onto the IO-pad.

THEOREM. *An* L*-machine of size* N *can process* MEMBER/INSERT/DELETE/XMIN *commands with a compute time of* $O(\log N)$ *and a period of* $O(1)$*, as long as the number of elements in the set remains* $\leq N$*. (It is assumed that* INSERT*s always add new keys and that* DELETE*s always remove existing keys.)*

Extraneous insertions (i.e., INSERT(k) commands with k already in the set) lead to severe problems, for they make processors shift their keys right while they shouldn't. Likewise extraneous deletions (i.e., DELETE(k) commands with k not in the set) make processors erroneously shift keys left, crushing the smallest key > k currently in the set. A possible way out suggested in [18] is to allow "holes" in the array, i.e., to accept that some processors in the initial segment of the array hold no key. In this way, an INSERT command could proceed as indicated (because it would merely create an additional hole) and the procedure for DELETEs could simply consist of the erasure of the key to be deleted. We only need to make sure that the number of holes doesn't grow out of hand, to endanger

e.g. the rapid answering of XMIN commands. We shall distinguish between empty locations at the right end of the array (marked with ∞) and embedded empty locations at the front (marked by *).

THEOREM. *An* L*-machine of size* N *can process* MEMBER/INSERT/DELETE/XMIN *commands with a compute time of* $O(\log N)$ *and a period of* $O(1)$*, as long as the number of elements in the set remains* $\leq N/2$. *(Extraneous insertions/deletions are allowed.)*

PROOF. The technique is due to Ottmann, Rosenberg and Stockmeyer [18] but we render it in a considerably simplified form. The set will be maintained such that the following invariants hold: (I1) the first processor is not starred, and (I2) every starred processor has a non-starred right neighbour.

Extraneous insertions can lead to two consecutive starred processors and (ordinary!) deletions to even three starred processors in a row, assuming a deletion simply "stars" the deleted key. Define an auxiliary command COMPRESS, which makes a processor shift its key left if the left neighbour is starred. To reinstate the invariant, it clearly is sufficient to let every processor right of the presumed location of the update do one or two COMPRESSes after having performed the regular steps for an INSERT or a DELETE command, respectively. An XMIN will always move the smallest key correctly out onto the IO-pad due to (I1), but the left shift generated for the entire array could bring (or rather: leave) a star in the first cell. Fortunately (I2) is preserved in a left shift and guarantees that the second processor is not starred. To maintain the invariant (I1) it thus sufficies to do one COMPRESS after every XMIN.

(I1) and (I2) imply that at most $\lfloor N/2 \rfloor$ processors may get (and remain) starred, thus reducing the effective capacity of the machine to $\lceil N/2 \rceil$ keys. □

Ottmann, Rosenberg and Stockmeyer [18] argue that the compute time for the given set of commands can be reduced to $O(\log N)$ by snaking the sorted chain of keys through the (upper) levels of the tree and creating a proper barrier to "bounce" signals back to the root.

4. SYSTOLIC ARRAYS

Until now we only saw designs in which data and instructions can enter through a single IO-port. One- and two-dimensional systolic arrays are

designed so vectors of data can enter into the computation simultaneously, by letting all processors on the boundary have IO-ports to the environment (see e.e. figure 4). In this Section we shall explore the potential of the parallel pipelining of data in a number of applications. We shall primarily discuss the problem of comparing tuples $(a_1,\ldots,a_N)$ and $(b_1,\ldots,b_N)$, but the underlying principles will extend to familiar problems like the systolic multiplication of two $N \times N$ matrices.

Let us suppose first that $(a_1,\ldots,a_N)$ is fixed and that we want to compare it to many tuples $(b_1,\ldots,b_N)$. A first design that comes to mind is shown in figure 4. It is an array of N processors, in which

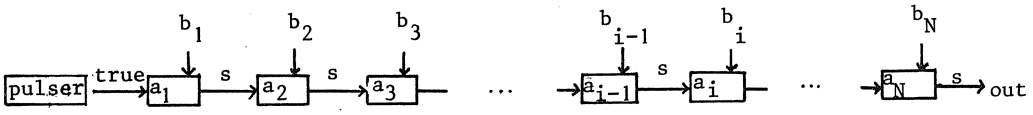


figure 4

the i^{th} processor is fixed to contain the i^{th} component of the tuple a. The tuple b is entered such that its i^{th} component is input to the same processor, so a comparison of a_i and b_i can take place. A signal s (starting out with value true) is chased from left to right to collect the results. The tuples match if and only if s still has value true

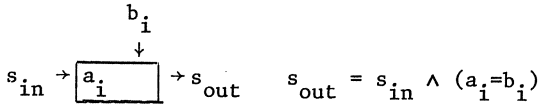


figure 5

when it reaches the right end of the array. Figure 5 shows the simple action of every processor.

The design of figure 4 appears to have a period of $O(N)$. Yet there is no reason to let the 1^{st}, 2^{nd},... processor idle as soon as the s- signal has passed. To take advantage of it, we shall modify the design and use a *skewed* input format (figure 6): for all $1 \leq i \leq N-1$ the datum b_{i+1} is input into the $(i+1)^{st}$ processor exactly one clock pulse after b_i is input to the i^{th}. The net effect is that the $(i+1)^{st}$ component is input

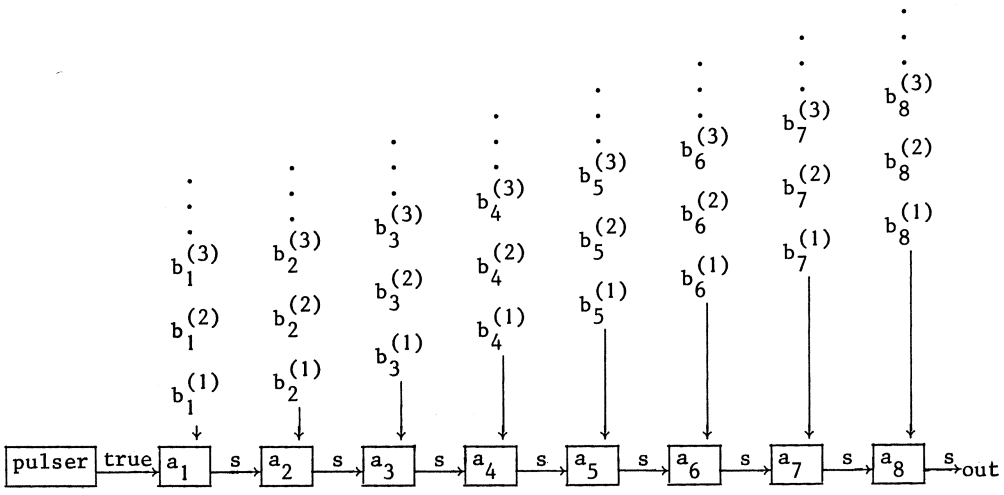


figure 6

just when the result signal s of the comparison between $(a_1,\ldots,a_i)$ and $(b_1,\ldots,b_i)$ comes in from the left. It should be clear that this design can be pipelined with a period of $O(1)$. The result of a comparison is available one clock pulse after entering the last component of a b-tuple.

THEOREM. *A linear systolic array of size* N *can do tuple comparisons with a fixed vector of lenght* N *with compute time* $O(N)$ *and period* $O(1)$*, using a skewed input format.*

It is interesting to note that there is nothing special about using the operations $\wedge$ and = in the processors (cf. figure 5). If we use operations + and . instead, s essentially accumulates the inner product of the tuples $(a_1,\ldots,a_N)$ and $(b_1,\ldots,b_N)$ as vectors. We thus have a systolic algorithm for a variety of problems that all conform to the same algebraic laws.

Now imagine that we have a "long" string $b = b_1b_2b_3 \ldots b_n$ $(n \geq N)$ and enter the tuples $(b_1,\ldots,b_N)$, $(b_2,\ldots,b_{N+1})$, $(b_3,\ldots,b_{N+2})$ and so on. The result is that $(a_1,\ldots,a_N)$ gets compared to every substring of b of lenght N, and the linear systolic array effectively becomes a *pattern matcher*. Closer inspection shows that it isn't really necessary to

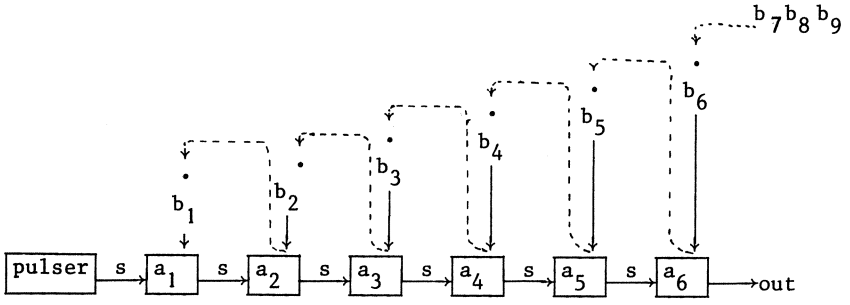


figure 7

enter all tuples separately. If we let each tuple lag by on additional clock pulse in time, then the next symbol required at the port of a processor can be sent over from its right neighbour and be received just when it is needed (see figure 7). Immediately after inputting the i^{th} symbol the array outputs a signal (s) that indicates whether the last N symbols match with $(a_1,\ldots,a_N)$ or not. Figure 8 shows the design we now

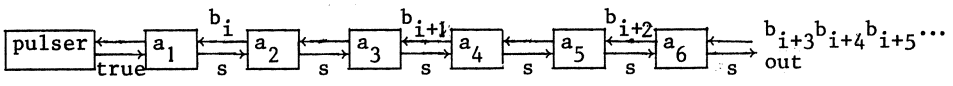


figure 8

have in its essential form. The string b is steadily moving through the array and the output indicates whether a match occurs or not. It is exactly the design of a *systolic pattern matcher* as given by Foster and Kung [8].

Now suppose we have a series of K tuples $a^{(k)} = (a_1^{(k)},\ldots,a_N^{(k)})$ and $b^{(k)} = (b_1^{(k)},\ldots,b_N^{(k)})$ and we wish to compare all couples $a^{(k)}$ and $b^{(k)}$, for $1 \leq k \leq K$. Returning to the design of figure 6 it is clear that a similar algorithm will do, provided we make sure that every processor is loaded with the proper $a_i^{(k)}$ at the right time to match a corresponding $b_i^{(k)}$. The solution is, of course, not to fix the a-symbol in a processor

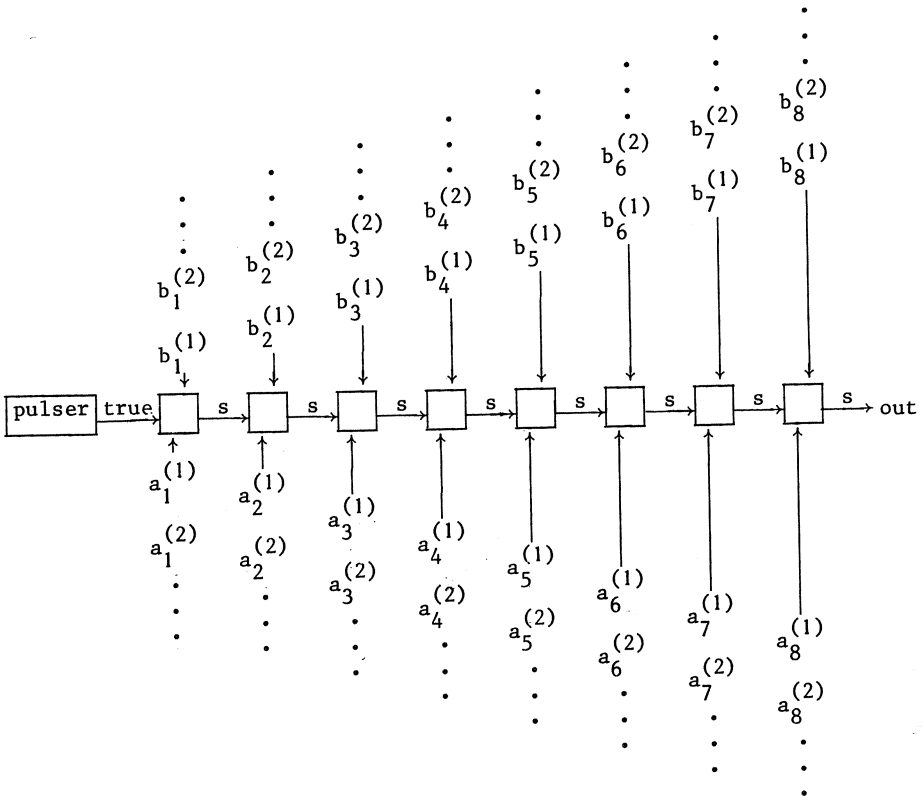


figure 9

but to feed in the a-tuples one after the other, in the same (but symmetrical) skewed manner as the b's. After reading in another pair of symbols $a_N^{(k)}$ and $b_N^{(k)}$ into the N^{th} processor, the array will output the result of comparing the entire tuples $a^{(k)}$ and $b^{(k)}$ in the cycle. The design is more appealing perhaps if we use the operations + and . instead of $\wedge$ and = in the processors.

THEOREM. *A linear systolic array of size* N *can compute inner product of pairs of vectors of length* N *with a compute time of* $O(N)$ *and a period of* $O(1)$, *using a skewed input format.*

The result is a classical one in pipelined computation, which thus holds in the current framework as well.

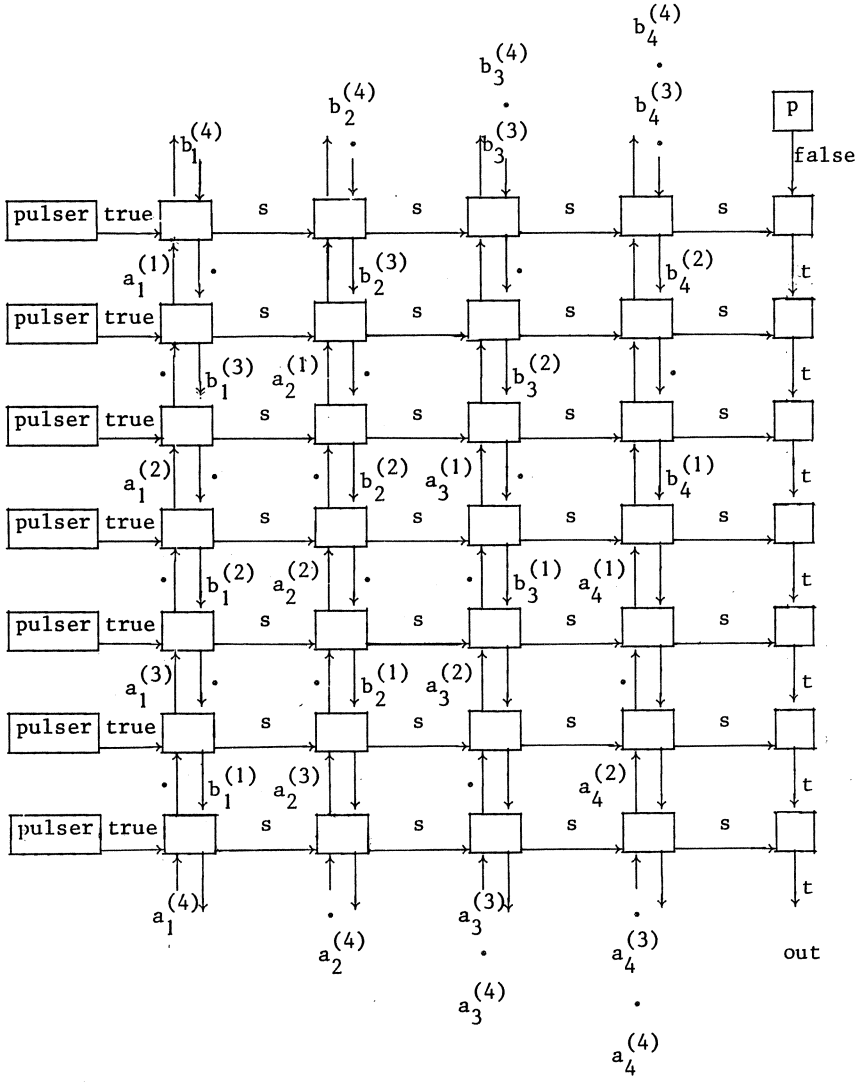


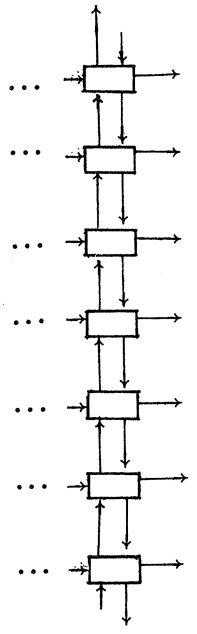
figure 10

Finally suppose that we wish to compare every a-tuple to every b-tuple and output signals which indicate with every (say) b-tuple whether it was matched at least once. In this way we would be able to select the tuples in the *intersection* of the two sets. The idea is, of course, to use a K-fold repetition of the linear design of figure 9 so each of the a-tuples comes across each of the b-tuples. A slight problem arises in the stepping pattern if we do so. For example, after $a_1^{(1)}$ and $b_1^{(1)}$ have met in a processor they both move and $a_1^{(2)}$ and $b_1^{(2)}$ immediately take their position. It means that $a_1^{(1)}$ and $b_1^{(2)}$ and $a_1^{(2)}$ and $b_1^{(1)}$ pass without meeting in a processor, i.e., without an apportunity to compare them! As shown in figure 10 the problem is easily solved by separating both the a-tuples

and the b-tuples by a "dummy" tuple (dummies are drawn as dots). The number of processor rows must be increased to 2K - 1 to make it work. To collect the results of the comparisons an additional column of processing elements is added to the right of the array. The t-signal sent down is "v" -ed with the s-signal coming in from every row. Observe that the t-signal steps along with the b_N-symbol of every $b^{(k)}$. By the time $b_N^{(k)}$ leaves the bottom right processor, the value of the t-signal that appears on the out-wire during the next clock period will indicate whether $b^{(k)}$ matched any of the a-tuples! The design has been suggested by Kung and Lehmann [12] for use in a relational database multiprocessor environment.

THEOREM. *A two-dimensional systolic array of size* $O(k)$ *by* $O(N)$ *can process two sets of* K *tuples of lenght* N *and determine their intersection in* $O(K+N)$ *time.*

An interesting result is obtained if we again replace the operations $\wedge$ and = in a processor by + and $\cdot$. In this case the design of figure 10 essentially computes all inner products $c_{k\ell} = a^{(k)} * b^{(\ell)}$ $(1 \le k, \ell \le N)$. If we omit the rightmost column and let each "s" -wire carry its value to output, then we get the inner products in the skewed order as



$$c_{14}$$

$$c_{24} \cdot c_{13}$$

$$c_{34} \cdot c_{23} \cdot c_{12}$$

$$c_{44} \cdot c_{33} \cdot c_{22} \cdot c_{11}$$

$$c_{43} \cdot c_{32} \cdot c_{21}$$

$$c_{42} \cdot c_{31}$$

$$c_{41}$$

figure 11

shown in figure 11. Now let K = N and interpret $a^{(1)}$ to $a^{(N)}$ as the rows of an $N \times N$ matrix A and $b^{(1)}$ to $b^{(N)}$ as the columns of an $N \times N$ matrix B.

The systolic array we designed exactly produces the coefficient of the product matrix C = A*B, with a compute time of $O(N)$!

THEOREM. *A two-dimensional array of size $O(N)$ by $O(N)$ can process two $N \times N$ matrices and output their product (as a matrix) in a compute time of $O(N)$. No (substantially) smaller array will do to get the same compute time.*

PROOF. It only remains to show that the $O(N^2)$ size bound of the systolic array cannot be substantially improved. This is a simple consequence of a theorem of Savage [20] that states that a matrix multiplier with area A and compute time T must satisfy $AT^2 = \Omega(N^4)$. □

In general the design can be applied to multiply $K \times N$ and $N \times K$ matrices. (See Katona [9] for some other uses of the design.)

5. A SYSTOLIC MATRIX INVERTER

In this Section we shall develop a systolic algorithm to invert an $N \times N$ matrix $A = (a_{ij})$ in time $O(N)$, using $O(N^2)$ processors. The algorithm is based on Gaussian elimination (see e.g. [6]) and assumes that no pivoting is needed during the process. It serves here to demonstrate the intricacies of pipelined computation. We shall first discuss the basic algorithm and modify the data movement in it to suit our purposes.

The Gaussian algorithm to compute A^{-1} (without pivoting) operates as follows. Extend A to a $N \times 2N$ matrix $(A\ I)$ by juxtaposing an $N \times N$ identity matrix and apply elementary transformations to it until a matrix $(I\ B)$ is obtained. Then $B = A^{-1}$. Permissible transformations are *rowmul*'s (multiply a row by a scalar) and *rowsub*'s (subtract a scalar multiple of a row from another row). The precise algorithm we use is due to Pease [9]:

```
for i := 1 to N do
   begin
        rowmul: multiply the i^th row by a_ii^-1;
                for j := 1 to N do
                   begin
                       if j ≠ i then rowsub: substract a_ji times the
                          i^th row from the j^th row
                   end
   end;
```

(In the algorithm a_{ii} and a_{ji} are used as variables and thus denote the values in the corresponding locations of A as they are at the time of reference.) Figure 12 shows how the algorithm works on a simple example.

$$\left(\begin{array}{ccc:ccc} 2 & 0 & 2 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \xrightarrow{(i=1)} \left(\begin{array}{ccc:ccc} 1 & 0 & 1 & \frac{1}{2} & 0 & 0 \\ 0 & -1 & 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \xrightarrow{(i=2)}$$

$$\left(\begin{array}{ccc:ccc} 1 & 0 & 1 & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & \frac{1}{2} & -1 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 1 & 1 \end{array}\right) \xrightarrow{(i=3)} \left(\begin{array}{ccc:ccc} 1 & 0 & 0 & \frac{3}{2} & -2 & -2 \\ 0 & 1 & 0 & \frac{1}{2} & -1 & 0 \\ 0 & 0 & 1 & -1 & 2 & 2 \end{array}\right)$$

figure 12

To obtain an algorithm within our present framework, we shall aim at having a processor in every "cell" (square) of the $N \times 2N$ matrix. The key to systolicism is to observe the necessary data movement, to make it regular, and pipeline it. Consider Gauss's algorithm for i = 1. To do the rowmul one could pass on the value of a_{11}^{-1} to the subsequent processors in the first row. Passing it on means: multiply, and send a_{11}^{-1} to the right neighbour. To do the rowsub for $j (j \neq 1)$ one must spread the value of a_{j1} through the row, and next get the values from the first row (that should sift down through every column) and do the required multiply (by a_{j1}) and substract (from the resident row element). It suggests that we better start by passing on the entire first column and temporarily store the values in an auxiliary field of the processors in every column. Only in the first row can we immediately do the necessary Gauss step right-away. Next we pass all (now modified) values of the first row down to the rows below and do the rowsub's, one after the other. This gives the algorithm the flavor of a cyclic (i.e., repeating) 2-phase process: send the first column right, and (next) send the first row down. For the sake of systolism it would be desirable if this cycle could be repeated for i = 2, i = 3,... . Observe that the first row is turned into a unit vector (cf. figure 12) and never again is the sight of much activity after the first step. Thus, while sending the first column right, we might as well move the column and let it "exchange" its way over to the end, thus effectively moving the

2^{nd}, 3^{rd},... column left one position. If we do a similar exchange with the first row (which puts it at the bottom af the matrix and moves up all other rows by one), then we have created a starting position as before (although effectively with i = 2 and e.g. a_{22} in the upper left corner). With this modification

$$\left(\begin{array}{ccc:ccc} 2 & 0 & 2 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \Rightarrow \left(\begin{array}{ccc:ccc} -1 & 0 & -\frac{1}{2} & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \Rightarrow$$

$$\left(\begin{array}{ccc:ccc} \frac{1}{2} & -\frac{1}{2} & 1 & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & -1 & 0 & 0 & 1 \end{array}\right) \Rightarrow \left(\begin{array}{ccc:ccc} \frac{3}{2} & -2 & -2 & 1 & 0 & 0 \\ \frac{1}{2} & -1 & 0 & 0 & 1 & 0 \\ -1 & 2 & 2 & 0 & 0 & 1 \end{array}\right)$$

⇒: "send" the first column right, next "send" the first row down.

figure 13

the algorithm has become completely regular and, as we shall see, amenable to pipelining. After N cycles of the 2-phase process all rows are back in their original order, but the columns are still halfway a full shift over 2N places. It means that the inverse now appears in the first rather than the second N × N block of the array and has effectively overwritten A, which is just as well. Figure 13 shows the modified algorithm applied to the example matrix of before.

As it stands each cycle of the algorithm takes $O(N)$ parallel moves and (thus) the entire routine takes $O(N^2)$ time, assuming the processors can be timed right to run it. Observe that each cycle of the algorithm consists of two "waves": one moving from left to right (doing a rowmul in the first row and spreading the a_{j1}'s in the lower rows) and a next

the wave from left to right:

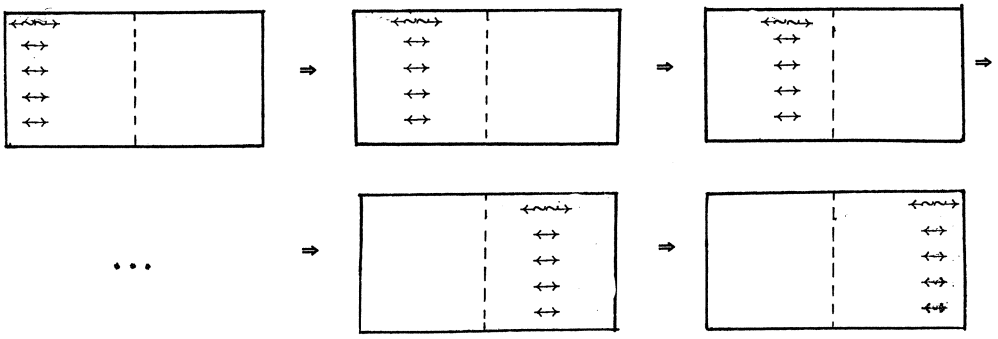


figure 14

the subsequent wave from up to down:

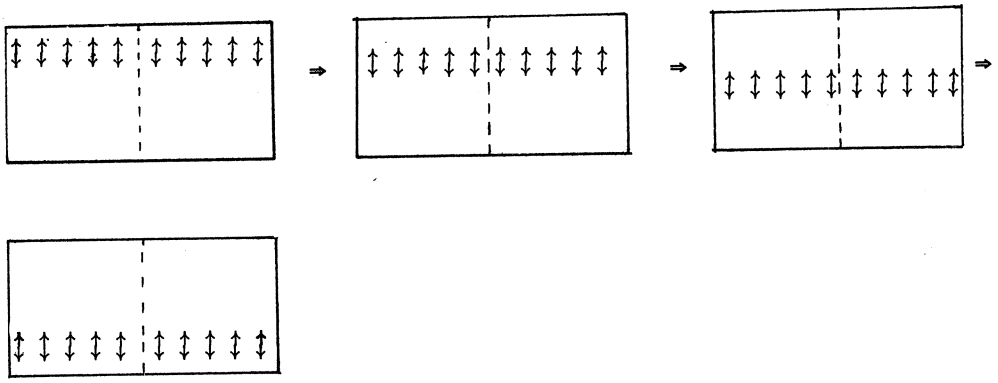


figure 15

one from "up" to "down" (exchanging the values of the first row downwards and executing the necessary rowsub's, using the a_{j1} values just deposited in the preceding wave). Figures 14 and 15 show how the waves progress, with "↔" and "↕" indicating where the "action" occurs. The wiggly "↜⇝" in the first wave indicates that a rowmul is carried out along the way.

Note that the down wave could begin at a processor as soon as the right wave has passed. And after a down wave has passed, the right wave of the next cycle can be started up. It follows that the regular wave pattern of the algorithm allows us to pipeline the computation and let the waves follow another at a short distance (only one or two clock periods). To achieve it we have to "skew" the wave fronts, so they indeed move completely in parallel fashion. Figures 14^{bis} and 15^{bis} show how this

looks like for separate waves, and figure 16 shows the

a skewed right wave:

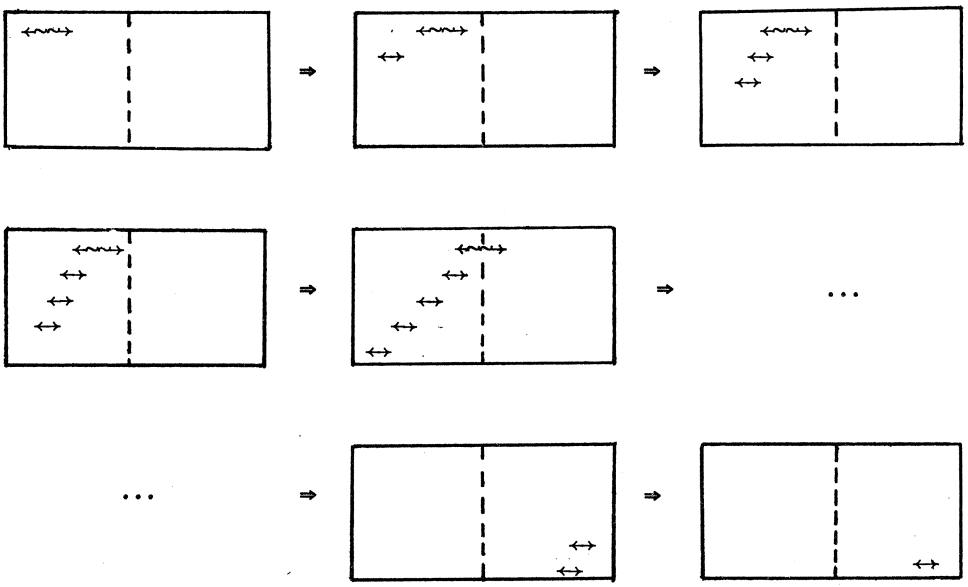


figure 14^{bis}

a skewed down wave:

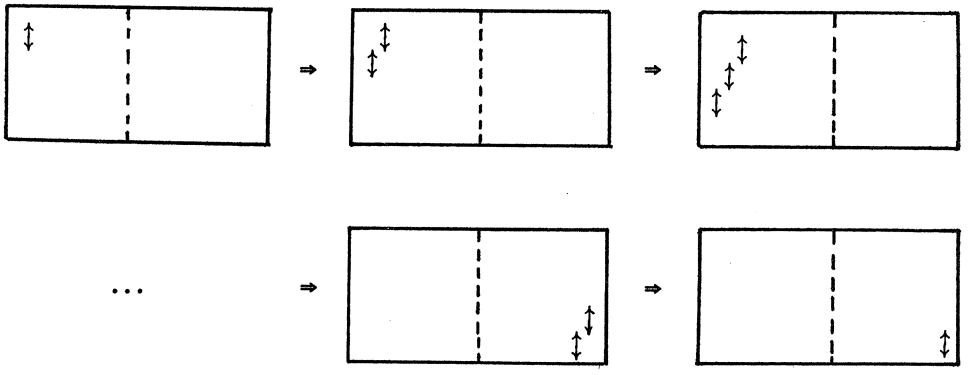
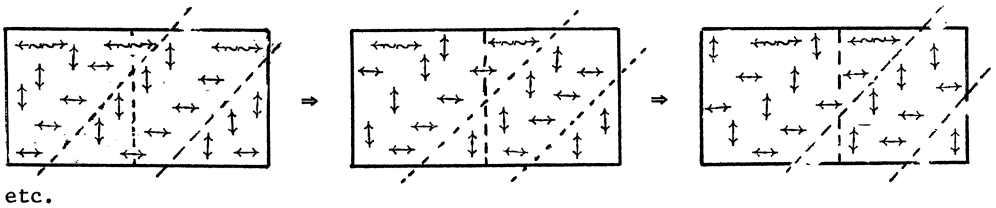


figure 15^{bis}

pipelining:



etc.

figure 16

waves of a cycle trailing (between ⋰ ⋱) across from the upper left to the lower right corner.

THEOREM. *A two-dimensional systolic array of size* N *by* 2N *can compute the inverse of a (stored)* N × N *matrix in* $O(N)$ *time using Gauss's algorithm (assuming no pivoting is needed).*

PROOF. Put a processor in every cell of the N × 2N matrix. Each processor will have two data registers and a periodic clock (or state indicator). The first (a-)register contains the value of the matrix element that is stored here, the second (b-)register contains the datum that is being passed on. The clock essentially cycles through four states, corresponding to the neighbour with whom information is exchanged (watch e.g. the continuous activity at a single cell in figure 16): ℓ ("get value from the b-register of the left neighbor"), r ("exchange"), u ("exchange") and d ("compute the rowsub"). We shall actually let the downward exchanges move through the a-registers. The processors along the boundaries will be considered separately as they miss some of the "neighbours" refered to. The activity of the array is started by sending a control signal immediately preceding the front of the very first cycle, which turns everybody's clock to ℓ.

For a concrete description of the actions of a processor we shall adopt Katona's method of specifying transitions through "configuration terms" ([9]). Processors will be placed in categories according to their location in the array (see figure 17): category I are all cells in the

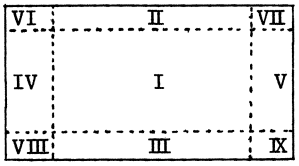
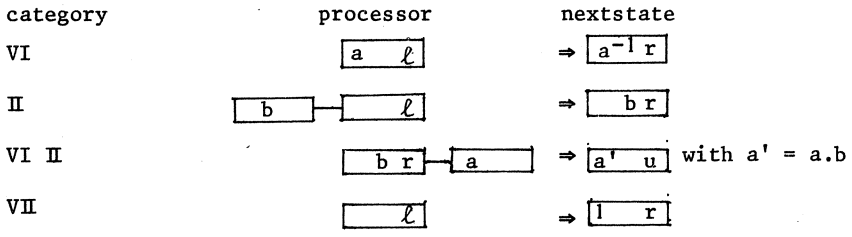


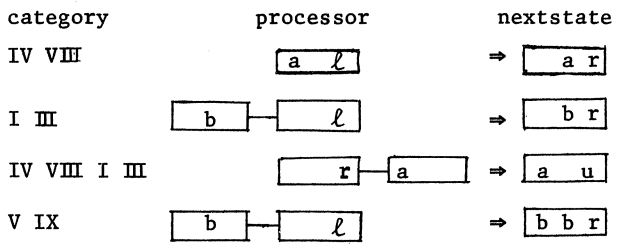
figure 17

interior, and categories II to IX are spread along the boundary. The transitions can be grouped as follows:

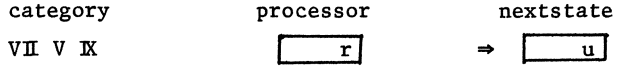
(i) send and exchange the multiplier through the first row (this implements the wiggly ⇜⇝)



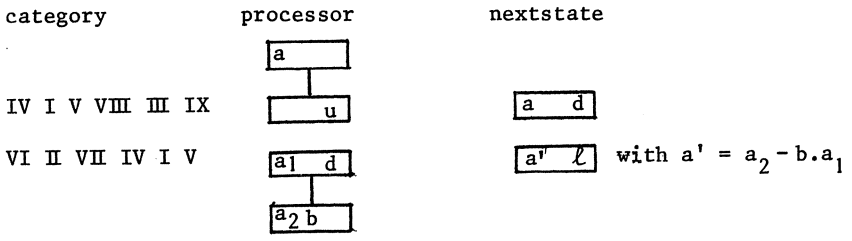
(ii) send and exchange the a_{j1} right (through the b-registers) as in the first wave of a cycle



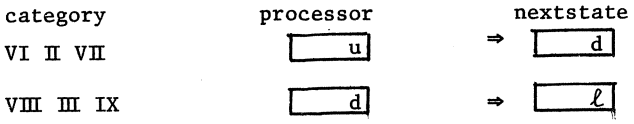
For the processors with no right neighbour we have, in addition:



(iii) send and exchange the elements of the first row down (through the a-registers) according to the second wave of a cycle



For the processors with no neighbour above or below we have, in addition:



The configuration terms only display the register contents that are of use in a transition. Unspecified ("empty") fields remain unaltered. The last wave of the final (N^{th}) cycle should carry a control signal that turns processors off (as they return to the ℓ state again).

Observe that N cycles of the algorithm thus trail over the array. The compute time is easily seen to be $O(N)$. □

A simple observation enables us to reduce the size of the processor array from N by 2N to N by N. (A related observation occurs in [19].) Returning to the non-pipelined version of the algorithm, consider the right

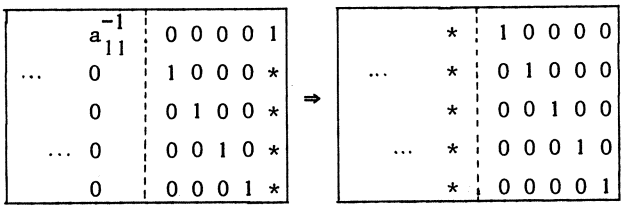
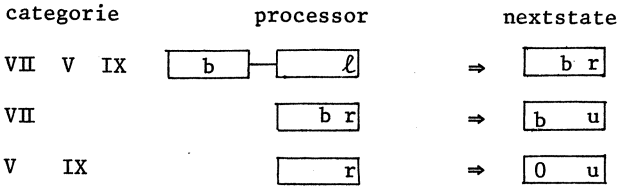


figure 18

N by N block after the first (left to right) wave of a cycle has ended and exchanged the first column towards the last position in the processor array (see figure 18). The next (downward) wave will turn this vector into a unit and exchange it towards the bottom position, thus effectively turning the entire right block into the identity matrix! It follows that we may as well eliminate the right block from the processor array, provided we let the processors along the (new) right boundary act as if the block was still there:



all other transitions remain as they were. The entire procedure of pipelining, of course, holds true for the curtailed array as well. There is no hope that the size of the array can be reduced to anything less than $O(N^2)$ if the linear processing time is to be maintained, in view of the results of Savage [20]. By noting (cf. [1], thm 6.8) that

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix}$$

it is not hard to see that the lowerbounds for matrix multiplication (cf. Section 4) apply to inversion as well. Hence, the systolic matrix inverter is essentially optimal with respect to the "AT^2" measure.

The design is interesting, for it proves that Gauss's algorithm contains a tremendous degree of parallelism. The most interesting part perhaps is the possibility to pipeline, which makes crucial use of the assumption that no pivoting is needed. The assumption is valid for e.g. symmetric matrices that are known to be positive definite (cf. [7]). If pivoting is required (for whatever reason, including numeric stability), then a third wave is added to every cycle which moves "backward" and

thus prevents the expedience of pipelining. Cycles still need $O(N)$ time, but the execution can no longer be overlapped. The $O(N^2)$ algorithm for matrix inversion that results still improves upon the fastest algorithms known in the sequential case. (This is Pease's result, cf. [19].)

It should take little effort to modify the systolic matrix inverter to a systolic "LU-decomposer". The underlying algorithm (again ignoring the need to pivot) is quite similar, but only does its rowsub's for $j > i$. After N cycles this produces the matrices U (upper triangular) and L^{-1} (lower triangular) in distinct portions of the processor array (see figure 19). Applying the inversion algorithm in the right block, the desired L

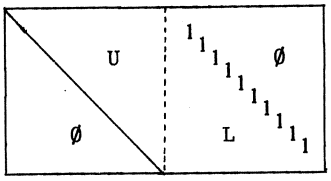


figure 19

matrix is obtained. In Kung and Leiserson [13] the LU-decomposition was produced slightly differently.

6. REFERENCES

(Reference [16] is not cited in the text.)

[1] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] ARMSTRONG, P.N. & M. REM, *A serial sorting machine*, preprint, Dept of Computer Science, California Institute of Technology, Pasadena, Cal., 1978.

[3] BENTLEY, J.L. & H.T. KUNG, *Two papers on a tree-structured parallel computer*, Tech. Rep. CMU-CS-79-142, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979.

[4] BHATT, S.N. & C.E. LEISERSON, *How to assemble tree machines*, Proc. 14th Annual ACM Symp. Theory of Computing, San Francisco, 1982, pp. 77-84.

[5] CODD, E.F., *Cellular automata*, Acad. Press, New York, NY, 1968.

[6] FADDEEV, D.K. & V.N. FADDEEVA, *Computational methods of linear algebra*, Freeman, San Francisco, Cal., 1963.

[7] FORSYTHE, G.E. & C.B. MOLER, *Computer solutions of linear algebraic systems*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1967.

[8] FOSTER, M.J. & H.T. KUNG, *The design of special purpose VLSI chips*, Computer 13 (1980) 26-40.

[9] KATONA, E., *Cellular algorithms for binary matrix operations*, in: W. Händler (ed.), CONPAR 81, LN-CS vol 111, Springer Verlag, Heidelberg, 1981, pp. 203-216.

[10] KUNG, H.T., *Let's design algorithms for VLSI systems*, Techn. Rep. CMU-CS-79-151, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh , Pa., 1979.

[11] KUNG, H.T., *Why systolic architecture*, Computer 15 (1982) 37-45.

[12] KUNG, H.T. & P.L. LEHMANN, *Systolic (VLSI) arrays for relational database operations*, Techn. Rep. CMU-CS-80-114, Dept of Computer Science Carnegie-Mellon Univ., Pittsburgh, Pa., 1980.

[13] KUNG, H.T. & C.E. LEISERSON, *Systolic arrays for (VLSI)*, Techn. Rep. CMU-CS-79-103, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979. (See also: [17], chap 8).

[14] LEGENDI, T., *Cellular algorithms and their verification*, in: W. Händler (ed.), CONPAR 81, LN-CS vol 111, Springer Verlag, Heidelberg, 1981, pp. 169-188.

[15] LEISERSON, C.E., *Systolic priority queues*, Techn. Rep. CMU-CS-79-115, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979.

[16] LEISERSON, C.E., *Area efficient VLSI computation*, Ph. D. Thesis, Techn. Rep. CMU-CS-82-108, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1982.

[17] MEAD, C.A. & L.A. CONWAY, *Introduction to VLSI systems*, Addison-Wesley Publ. Comp., Reading, Mass., 1980.

[18] OTTMANN, Th., A.L. ROSENBERG & L.J. STOCKMEYER, *A dictionary machine (for VLSI)*, Rep. RC9060 (#39615), IBM TJ Watson Research Cntr., Yorktown Heights, NY, 1981.

[19] PEASE, M.C., *Matrix inversion using parallel processing*, J. ACM 14 (1967) 757-764.

[20] SAVAGE, J.E., *Area-time trade offs for matrix multiplication and related problems in VLSI models*, J. Comp. Syst. Sci. 22 (1981) 230-242.

[21] SONG, S.W., *On a high-performance VLSI solution to database problems*, Ph.D. Thesis, Techn. Rep. CMU-CS-81-142, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1981.

VLSI, ALGORITHMS AND GRAPHICS *

D.P. Dobkin
Princeton University, Princeton, USA

ABSTRACT

Practical and theoretical aspects of the VLSI design process are discussed. The technology is described, algorithms for design automation are considered and details of two actual designs are presented.

1. INTRODUCTION

Within the next few years, the technology will exist to produce integrated circuits composed of a million or more transistors. Very Large Scale Integrated (VLSI) circuits of these sizes will revolutionize the nature of computation if the possibilities of productive design and use of such circuits can be realized, Numerous problems of mathematics, theoretical and practical computer science, engineering and physics must be more fully understood if we are to successfully apply the breakthroughs which allow the fabrication of transistors of micron and sub-micron size. The excellent book of Mead and Conway [MC80] represents a strong first step in this direction. Although all the assumptions in this book may not be preserved under arbitrary scalings and significant miniaturization, we shall follow their presentation in what follows here.

In this presentation, we discuss some practical and theoretical issues involved in VLSI design. Our discussion considers issues involved in the design of algorithms for the automation of the VLSI layout process. Practical issues of design encountered during the creation of two actual VLSI designs are also presented. As such, the details of current VLSI design

*)This research supported in part by the National Science Foundation under grant MCS81-14207.

systems are considered as they were used in the creation of these designs. While the two designs considered are of a relatively basic nature, a 16-bit wide divider and 16-bit wide adder, they are of sufficient complexity to give detailed insight into the design process. Furthermore, the first design has been successfully fabricated and tested. Within this discussion, we also consider more sophisticated designs composed of building blocks of the type discussed here. In particular, designs of VLSI engines for geometric computing are considered and the components that one might want to consider in constructing a geometry engine on a chip are discussed.

The components of this paper are threefold, as the title suggests. In the next section, we discuss VLSI and the VLSI implementation of two circuits in different design environments. Section 3, devoted to algorithms, presents a consideration of two algorithmic processes, geometric in nature, involved in VLSI design systems. Finally, in the graphics section, the nature of one aspect of a VLSI geometry engine is discussed.

This paper is intended to briefly survey the topics mentioned here. No attempt is made to be complete. Instead, references are included and the interested reader is referred elsewhere for further details.

2. ASPECTS OF THE VLSI DESIGN PROBLEM

The excellent book of Mead and Conway [MC80] made MOS design accessible to a larger group of computer scientists than had been previously possible. As such, new problems of interest to theoretical computer scientists have been proposed and new research areas are growing. One of the most interesting problems is the development of design aids for VLSI design. While problems of computer aided design date back to the earliest computers, there had been little algorithmic analysis of methods of solution of such problems. In some sense, the recent flurry of research activity in this area can be viewed as a reconsideration of old problems and solutions in a new light. While the dimensionless design system provided by Mead & Conway may introduce assumptions which do not have universal application, it does provide a starting point from which design aids may be developed and circuits may be designed.

During the past year, I have had the opportunity, in collaboration with Scot Drysdale, to design two circuits using different approaches to the design process. Experiences with these designs are used here as a basic of a

discussion of design systems and their relative merits. Algorithmic aspects of these design systems form the basis of the next section.

The two circuits to be considered will be a 16-bit wide divider and a 16-bit wide adder. The divider was designed at Xerox PARC using graphics based design tools [FR79, FN81]. The adder is being designed at Princeton as part of a larger project. The adder is being built using procedural design tools [LSV82].

Our results are not meant to be a comparison or evaluation of the design system as much as a report on the design experiences in each environment. Indeed, the design systems are at different stages of development and the execution of designs in each case has thus far been brought to a different stage of completion. The goal here is to report of the powers and limitations of these different design systems and to consider the parts of each approach which are necessarily highly algorithmic in nature. In the next section, we explore algorithmic approaches which have been applied to two important geometric problems which arise in these contexts.

2.1. Graphics based design of a divider

The divider to be described herein was designed as a first exposure to VLSI design. The project was chosen as one of reasonable size whose execution could be completed in the available time while giving sufficient exposure to the design process. The division problem was stated as

$$\begin{array}{r} q \\ x \;\big\lfloor\; y \\ \hline r \end{array}$$

That is, given dividend $y = y_1 \ldots y_{16}$ and divisor $x = x_1 \ldots x_{16}$ find quotient $q = q_1 \ldots q_{16}$ and remainder $r = r_1 \ldots r_{16}$ such that $x = yq + r$ and $0 \leq r < y$. The computation of the quotient and remainder is done by repeated comparison and subtraction involving partial dividend (p) and divisor (x). After each cycle, a new bit (n) is shifted into the dividend and a quotient (q) is generated following

```
if p > x
    then
        p <-- 2*(p-x) + n
        q <-- 1
    else
        p <-- 2*p + n
        q <-- 0
```

This computation is done in a highly parallel manner. The comparison $p > x$ is done while the difference $p - x$ is computed (indeed as a by-product of the computation). Based on the value of $p > x$ (i.e. of q), either $p - x$ or p is shifted left by 1 bit and the new bit n shifted into the low order position of p.

The subtraction/comparison are done by a "borrow-look ahead" subtractor of design analogous to "carry-look ahead" designs used for adders [S80]. This construction will be described in greater detail for the adder below, so the details are omitted here. The controlling logic of the circuit consists of a value START read from an input pad and a value DONE read to an output pad. START = 1 initializes the circuit, starting the clock and enabling the buffered I/O pads to accept external inputs. The highest order bit of the quotient (actually q_0) is set to 1. After 1 cycle, the pads are set to accept output from the circuit and 16 subtract/compare-shift/generate quotient bit cycles follow. During each cycle, q_0 is shifted towards the DONE pad. After 16 cycles, the value of DONE is 1 and processing is completed.

A floor plan of the complete circuit is shown in Figure 1. The upper pads are used to input the dividend and output the (partial and complete) quotients. Shift registers here handle the shifting of bits. The lower pads are used to input the divisor and output the (partial and complete) remainders. A memory here stores the value of the divisor. The major portion of the design effort involved creating the subtractor/comparer. The major portion of the circuit area is comsumed by pads. Indeed, the chip dimensions (fabricated at $\lambda = 2.5\mu m$) are 8mm × 3mm with the subtractor involving 6mm × 1mm or 25% of the area. The design involves about 2500 transistors

and required about 3 man months of effort involving about 2 man months of design and 1 of debugging and testing. The tools used in the design were ICARUS [FR79] for graphical layout and MAGIC [FN81] for testing and simulation. Two fabrications of the chip were done due to difficulties in the first process. Of the six chips generated, one was successfully tested at a clock cycle time of approximately 10μs. This time was limited by the testing tools. We believe that the actual circuit can be clocked at a much faster rate.

Various lessons were learned from the process of designing, debugging and testing this circuit. Our inexperience with circuit design and hardware considerations led us to a mixed top-down/bottom-up method of design. The floor plan was modified as the circuit progressed and we had little insight into sizes of parts of the circuit until they were designed. Some good estimates and lucky guesses led to a final design which is nearly rectangular and (we believe) relatively compact. During the design, a graphics system proved to be a useful tool with its "what you see is what you get" structure. We were overzealous in λ-squeezing which proved harmful in later debugging of the circuit. Often, wires were twisted to save area rather than settling for a cleaner design.

During debugging, the design aids were surprisingly slow. A design rules check required 30 minutes on a very high speed processor. When a genuine error was found, it required a massive rewiring of the circuit. The graphics system was especially ill-suited to this task and 4 man hours were required to rewire 16 output pads into the circuit. After the circuit had been successfully design rule tested, during static simulation it was discovered that a complemented signal was needed in place of a signal in the design. This observation had drastic consequences since the affected signals occurred within the most densely designed portion of the circuit and rearrangement involved significant rewiring. Within our graphics based system, there was no facility for stretching wires across a horizontal (or vertical) line to allow new signals to pass. Such a change would have had significant impact on our design process.

The major algorithmic bottlenecks we felt in the design under the graphics-based paradigm were two - the need for improved routing tools and the need for more efficient design rules checking. The geometric aspects of these problems are discussed in further detail below. The design process gave us a greater respect for the complexity of VLSI design as well as

deeper insights into "bottom-up/top-down" design methodologies for design.

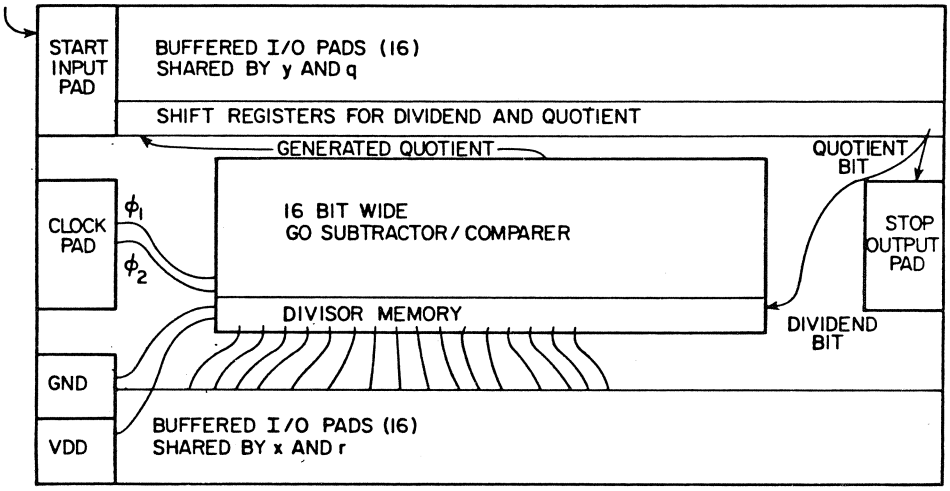


Figure 1. Divider floor plan

2.2. Constraint-based design of an adder

Having successfully tested the divider, attention was turned to the design of VLSI circuits for graphics applications. These applications are presented in greater detail in Section 4. As a first step, it was clear that a circuit to compute the sum of two integers would be necessary to most graphics designs. Attention was focused on this problem in order to gain insight into the design tools being developed at Princeton.

The goal was to design an efficient carry-look ahead adder in VLSI. The problem was modelled as

x	$x_1 \cdots x_{16}$
or	
y	$y_1 \cdots y_{16}$
---	---
z	$z_1 \cdots z_{16}$

As intermediate steps, $p_{i,j}$ and $g_{i,j}$ (for propagate and generate) primitives were defined with the properties that $p_{i,j}$ $(i \leq j)$ was true iff a carry would be propagated over the block of bits $i \ldots j$ and $g_{i,j}$ $(i \leq j)$ was true iff a carry would be generated over that block of bits. These quantities could then be computed as

$$p_{i,j} = x_i \vee y_i$$
$$p_{i,j} = p_{i,k} \wedge p_{k+1,j} \quad (i \leq k < j)$$
$$g_{i,j} = x_i \wedge y_i$$
$$g_{i,j} = (p_{i,k} \wedge g_{k+1,j}) \vee g_{i,k}$$

When the context permits, p_i (resp. g_i) will be used for $p_{i,j}$ (resp. $g_{i,j}$).

We note that the result $z_i = x_i \oplus y_i \oplus g_{i+1,16}$ and organize the computation as a series of levels from which the design can proceed. Level i may depend on results from both level $i - 1$ and $i + 1$ so proper timing of the circuit is necessary. The computation can be modelled by the tree of Figure 2a and the equations below. We make use of the two identities

$$A \oplus B = AB \vee \overline{A}\,\overline{B}$$
$$x_i \oplus y_i = \overline{p}_i \vee g_i$$

Level 0. Compute

$$z_i = x_i \oplus y_i \oplus g_{i+1,16} \quad i = 1, \ldots 16$$

Level 1. Compute

$$\overline{p}_i = \overline{x_i \vee y_i} \quad \overline{g}_i = \overline{x}_i \overline{y}_i \quad i = 1, \ldots 16$$

Level 2. For i even, compute

$$\overline{p}_{i-1,i}, \overline{g}_{i-1,i}, \overline{g}_{i-1,16} \quad \text{from} \quad \overline{p}_{i-1}, \overline{g}_{i-1}, \overline{p}_{i}, \overline{g}_{i}, \overline{g}_{i+1,16}$$

Level 3. For i a multiple of 4, compute

$$\overline{p}_{i-3,i}, \overline{g}_{i-3,i}, \overline{g}_{i-1,16} \quad \text{from} \quad \overline{p}_{i-3,i-2}, \overline{g}_{i-3,i-2}, \overline{p}_{i-1,i}, \overline{g}_{i-1,i}, \overline{g}_{i+1,16}$$

Level 4. For i a multiple of 8 (i.e. i=8,16), compute

$$\overline{p}_{i-7,i}, \overline{g}_{i-7,i}, \overline{g}_{i-3,16} \quad \text{from} \quad \overline{p}_{i-7,i-4}, \overline{g}_{i-7,i-4}, \overline{p}_{i-3,i}, \overline{g}_{i-3,i}, \overline{g}_{i+1,16}$$

Level 5. For i a multiple of 16 (i.e. i=16), compute

$$\overline{p}_{i-15,i}, \overline{g}_{i-15,i}, \overline{g}_{i-7,16} \quad \text{from} \quad \overline{p}_{i-15,i-8}, \overline{g}_{i-15,i-8}, \overline{p}_{i-7,i}, \overline{g}_{i-7,i}, \overline{g}_{i+1,16}$$

We adopt the convention that $g_{17,16} = 0$ and note that $p_{1,16}$ and computations leading to its value are unnecessary and can be removed. However, the nature of the circuit is such that their inclusion does not increase the layout area and does simplify the design process.

We propose to make the design such that all level i cells are identical for $i > 1$. Levels 0 and 1 may be viewed as a single cell organized to have width half that of a level 2 cell leading to a floor plan as shown in Figure 2b.

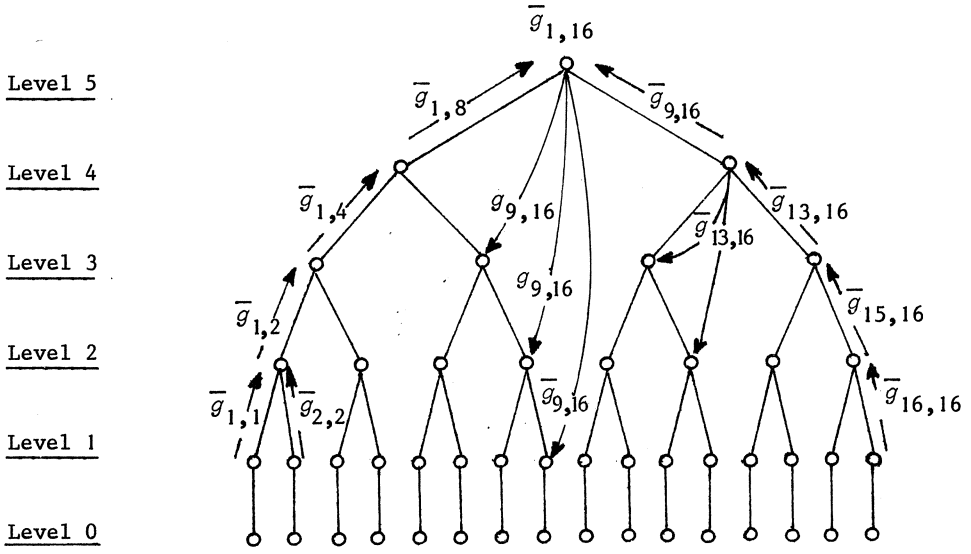


Figure 2a. Upward flows are $\overline{g}_{i,j}, \overline{p}_{i,j}$ where $j = i + 2^{k-1}$ for the flow from level k to level $k + 1$. Downward flows are $\overline{g}_{i+1,16}$ where i is a multiple of 2^k for the flow from level $k + 1$ to level k.

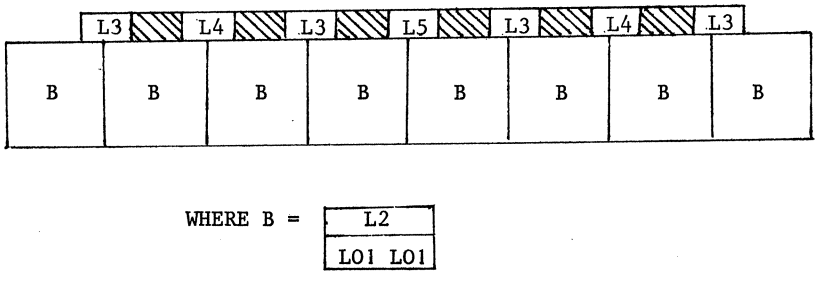


Figure 2b. The floor plan where L_i is a level i block. Crossed areas on the top level are used for routing of signals among the L3,L4,L5 blocks.

This design is currently being implemented in ALI [V82] which is a procedural design language. Within this language, wires are specified as boxes with relative positions (left, right, above, below). The specification is done by writing a PASCAL-like program which is converted by the ALI system into a system of linear constraints of the form

$$x_i \geq x_j + b$$
$$x_i = x_j$$

allowing for linear time solution of the constraints. The advantage of this approach is that a change in design results in simple changes to the underlying ALI program followed by recompilation. The changes which arose during the debugging phase of the divider would be greatly simplified at the expense of regenerating and solving all constraints mechanically.

Initial experience suggests that ALI will behave similar to the graphical tools described above for initial layout of cells. The "see what you get" feature of graphics-based system is an advantage over the one dimensional nature of ALI. Furthermore, ALI layouts are initially likely to consume more area. We roughly estimate this at 30-50%. However, the debugging of an ALI layout will be more pleasant and the nature of the language is such that a designer can hand optimize relevant cells on a working design. As such, it lends to a more suitable use of the "bottom-up/top-down" design paradigm which seems to arise in VLSI design projects. Further details on the comparison of graphical and procedural systems will appear in [DD82].

3. ALGORITHMS FOR GEOMETRIC PROBLEMS RELEVANT TO VLSI

As we observed in the previous section, two of the processes involved in design automation are routing and design rule checking. We model the later problem as that of intersection testing. Each of these problems is then of a geometric form. In this section we briefly survey methods of attacking these problems.

3.1. Geometric intersection problems

The design rules checking problem may be translated to problems on a set of rectangles representing the wires in a design. For example, let $R = \{r_1, r_2, \ldots, r_k\}$ be a set of rectangles, we may wish to ask queries of the form:

find the nearest neighbor of each r_i in R
find all intersections of elements of R
determine if any two rectangles of R intersect
count the intersection in R.

In the case of points, the Voronoi diagram [SH75] provides a useful data structure for considering such points. Further work on data structures recording history [DM80] allows the points of R to be preprocessed so that queries regarding new points are easily answered.

Our presentation here is limited to a consideration of the data structures used for reporting all intersections among a set of n line segments. We consider the segments $S_i = \overline{p_i q_i}, i = 1,\ldots n$ where $p_i = (x_i^s, y_i^s)$ and $q_i = (x_i^s, y_i^s), i = 1,\ldots n$, with $x_i^s \leq x_i^s$. We define $s_i(t)$ to be the intersection of s_i with the line $x = t$ with $s_i(t)$ undefined if t does not lie in the interval $[x_i^s, y_i^s]$. This notation leads to an order relation $<_x$ with $s_i <_x s_j$ if both are defined and $s_i(x) < s_j(x)$. $\leq_x$, $>_x$, $\geq_x$ and $=_x$ are similarly defined. This order property leads to the following theorem on which our algorithm is based:

THEOREM. *If s_i and s_j intersect, then there exist x_1 and x_2 such that $s_i \leq_{x1} s_j$ and $s_j \leq_{x2} s_i$.*

To derive an algorithm, we now define the set X as the set of end points, so $X = \bigcup_{i=1}^{n} (x_i^s \cup x_i^e)$. For each point x of X, we would like to maintain $\{s_i(x)\}$ in order (for all relevant i) so that testing can be easily done by noting where the order changes. Our algorithm [BO79] involves inserting elements into X as intersections are found (at an x to the right of our current position) and maintaining a balanced tree of $\{s_i(x)\}$ for each x of X in turn.

```
Initially sort X
    Maintain balanced tree (initially empty) ordering {S_i(x)} for x ε X.
    At each x of X (in increasing order)
        if x is a start vertex of s_k
            insert s_k(x) into the tree
            test to determine if s_k intersects either of its neighbors
                if so, add the intersection point to X and report the
                intersection.
        else if x is an end vertex of s_k
            insert s_k(x) into the tree
            test to see whether the neighbors of s_k intersect
                if so, add the intersection point to X and report the
                intersection.
```

else (x is the intersection point of segments m and n)
reverse $s_m(x)$ and $s_n(x)$ (assume w.l.o.g. that now $s_m(x) > s_n(x)$)
check m for intersection with the segment above it
check n for intersection with the segment below it
in each case, if an intersection is found, report it and add the point to X.

The running time of this algorithm can be expressed as $O(|X| \log n) + O(n \log n)$ with the first term giving the time for maintaining the tree and the list X and the second giving the time for initially sorting X. If s is the total number of intersections, this reduces to $O((s+n) \log n)$. For sparser intersections, this is considerably better than the $O(n^2)$ required by the naive algorithm. In a design rule checking context, we would expect this to be the case (or possibly abort after a fixed number of errors had been reported). However, in general other algorithms perform better than this "nearly optimal" algorithm. Recent work [KR82] explores this situation.

3.2. Routing geometries

A second tool which would facilitate the designer's task is an automatic router. Research in routing algorithms dates back to algorithms for laying out printed circuit boards in the early 1960's (see [L80] for a survey). Recent work has considered NP-completeness results as well as algorithms for problems solvable in polynomial time. We present here two techniques which have been applied to instances of routing problems. For a general statement of the problem, we consider two sets of ports $\{u_1,\ldots,u_n\}$ and $\{v_1,\ldots,v_n\}$ and ask for a routing which connects u_i to v_i $(1 \le i \le n)$ satisfying the conditions:

a) minimum separation is satisfied*
b) the routing minimizes
   i) total wire length
   ii) layout area (i.e. separation of components)
c) all connections are
   i) line segments and arcs of circles of constant radii
   ii) line segments at fixed angles
   iii) horizon and vertical line segments

Tompa [T80] proposes an exact solution to the problem a, bi), ci). We

*Throughout we assume that wires have no thickness and the minimum separation is 1.

assume that each set of ports is collinear and the two lines of ports are parallel separated sufficiently to allow a layout. The total wire length is to be minimized and wires may be line segments at any orientation or arcs of circles of constant radii. We say a layout is *good* if for all i, the wires connecting its i^{th} ports lie outside the circles of radius $|j-i|$ centered at u_i and v_j, $1 \le j \le n$. A layout algorithm then follows from the observations that a necessary and sufficient condition for a layout to be optimal is that it is good. Necessity follows easily from the problem statement. Sufficiency follows from the following observations about convexity.

DEFINITION. Let P be a continuous curve, p a point of P dividing P into curves P_1 and P_2. If q is a point not on P, then P is convex towards q at Q if and only if for all neighborhoods N of p, there are points p_i on $N \cap P_i$ $(i=1,2)$ such that $\overline{p_1p_2}$ does not intersect $\overline{pq}$.

THEOREM. *If P and Q are continuous non-intersecting curves of closest distance D and $p \in P$, $q \in Q$ have $d(p,q) = D$, then one of the conditions* a)-e) *holds*

a) *p is an endpoint of P*
b) *q is an endpoint of Q*
c) *P is convex towards q at p*
d) *Q is convex toward p at q*
e) *there exist neighborhoods of p and q within which P and Q are segments of parallel lines.*

Furthermore, there exist $p_0 \in P$ and $q_0 \in Q$ with $d(p_0,q_0) = D$ such that p_0 or q satisfies one of a)-d).

While this derivation provides an elegant characterization of solutions to the problem, it provides less elegant solutions. We may apply the algorithm to any pair of ports u_i,v_i to obtain in time linear in n a minimal length routing. However, quadratic time is required to process n pairs of ports because each segment might have n segments which need to be computed. Furthermore, the assumptions concerning wires at arbitrary angles and wires which are arcs is unrealistic for most design automation systems. Finally, the quadratic equations needed to describe circles may lead to the necessity of solving equations of high degree. Nonetheless, an algorithm exists to implement this procedure and provides insight into geometric characterizations of routing-like problems. The reader is referred to [T80] for details of this algorithm.

A second routing problem with an elegant solution is that which we characterize as a),b)ii),c)iii) where ports again lie on parallel channels which we want to separate by minimum distance while routing on only two layers (i.e. in only the horizontal and vertical directions). Two quantities have been defined by which such problems may be characterized. We consider the ports $\{u_i\}$ and $\{v_i\}$, $1 \le i \le n$ where u_i occurs at U_i and u_i at V_i along the relevant line. We assume the lines are horizontal and define the channel density at horizontal point h to be the number of wires which must cross that point. That is,

$$|\{i | u_i \le h \le V_i \text{ or } U_i \ge h \ge V_i \text{ but not } U_i = h = V_i\}|.$$

Clearly, this quantity will be a lower bound to the separation. However, as Dolev, et al [DKSSU81] note, there are problems of small channel density having layouts requiring a large separation. Consider, for example, the situation where $U_i = i$, $V_i = i + 1$, $1 \le i \le n$. To circumvent these difficulties, they define the conflict number W(i,j) by the table

	i < j	$i = j$	i > j
$U_j - V_i > j - i$	0	1	i-j+1
$U_j - V_i = j - 1$	j-i+1	0	o-j+1
$U_j - V_i < j - 1$	j-i+1	1	0

It is easily seen that in all but trivial designs, the conflict number is no less than the maximum channel density. By a greedy procedure, the layouts of separation equal to the conflict number may be realized yielding

THEOREM. *Minimum separation requires a number of channels exactly equal to the largest conflict number.*

PROOF. See [DKSSU81].

Again, the geometric elegance of this algorithm hides some difficulties which might arise in practice. We use this algorithm after components have been layed out to connect ports on different components. The algorithm then yields a layout of these components having minimum separation. A more integrated approach might combine the positioning of the ports during

design with a routing algorithm of this sort as well as allowing ports to occur on all edges of a component.

4. GRAPHICS

Our final topic involves the practical application of VLSI design systems of the type we have considered to problems of computer graphics. The display of images on a raster scan device requires substantial computing resources of a very specialized nature. We focus attention here on the simple problem of line drawing, undoubtedly the most basic problem in this area.

A raster display consists of a rectangular array of pixels. For each pixel location, associated memory maintains a value and the monitor displays light relative to this value. For a simple device, the monitor is capable of only bilevel (on/off) displays and a pixel is represented by a single bit of information. Grey scale monitors allow for pixels to be shaded. For example, 8 bits of memory allow the representation of 256 different shades of grey pixel. Color is produced by considering "grey scales" for the primary colors (red, green, blue) independently.

A line drawing algorithm plots a set of points $\{(x_i,y_i),(i=1...n)\}$ which best approximate a line. For a line in the first octant, i.e. at angle less than 45°, this is done by determining for each x_i on the line, the closest y_i such that (x_i,y_i) lies on the line. We do this in an iterative fashion from the minimum to maximum x - coordinatie of the line. Bresenham [B65] proposed the following algorithm to draw the line from (0,0) to $(x,y)(y\leq x)$.

```
xt := 0;
yt := 0;
setpixel (xt,yt);
slope := 2*y-x;
error := slope
incl := 2*(y-x);
inc2 := 2*y
for  xt := 1 to x do
     begin
         if error > 0 then
              begin
                  yt := yt + 1;
                  error := error + incl;
```

```
        end
    else
        error := error + inc2;
    setpixel (xt,yt).
        end;
```

The inner loop of this algorithm is executed many hundreds of times for each line drawn. In the algorithm statement, this loop has been made as tight as possible involving a compare to zero, an add and possibly an increment by one. It would be a simple matter to implement this algorithm in VLSI as part of a larger graphics engine. To do so, the comparison, the alternative adds and the incrementing by one would be done in parallel. At the time when all results were computed, the result of the comparison would be used to determine the new values of error and yt.

A difficulty of line drawing on a bi-level display is that honest representations of lines are not presented. A line of length 100 will be represented by turning on 100 pixels if horizontal but only 71 if diagonal. This difficulty is called aliasing. Various anti-aliasing schemes have been proposed. The key here is to view a line segment as the parallelogram bounded by parallel segments one half pixel above and below the given segment. A pixel is now viewed as a square of area 1. Using grey scale, the fraction of a pixel which is covered by the rectangle to the fraction which is lit. Under this scheme, lines are represented more accurately at added computational cost. A careful analysis of this problem by Field [F82] yields an algorithm whose inner loop involve 9 adds and 2 comparisons in the worst case where 3 grey scale values are set. This algorithm will lead to a simple VLSI implementation.

Extensions of this problem involving the careful drawing of triangles and 3 dimensional figures. A more difficult problem is that of performing transformations on objects accurately. At present, this is done by using floating point arithmetic with the result that "off-by-one" errors are common. Indeed, the rotation of a triangle by 45° eight times under most algorithms will not yield the original triangle on a bi-level display. We believe that more complex algorithms, which can be made to run sufficiently rapidly in VLSI, can alleviate this problem.

5. CONCLUSIONS

We have considered VLSI design systems, geometric algorithms and computer graphics in these lectures. Each of these topics is rich enough to provide many interesting lectures. Of necessity, we have highlighted a few ideas in each topic with the hope that the reader's appetite will be whetted to explore further.

6. REFERENCES

[B79] BENTLEY, J.L. and T. OTTMANN, *Algorithms for reporting and counting geometric intersections*, IEEE Transactions on Software Engineering, 5 (4) 1979, pp. 333-340.

[B65] BRESENHAM, J.E., *Algorithm for computer control of digital plotter*, IBM Systems Journal, 4 (1) 1965, pp. 25-30.

[DD82] DOBKIN, D.P. and S. DRYSDALE, *Design experiences in two environments*, to appear.

[DM80] DOBKIN, D.P. and J.I. MUNRO, *Efficient uses of the past*, Proceedings of 21st FOCS Symposium, Syracuse, New York, October, 1980, pp. 200-206.

[DKSSU81] DOLEV et al, *Optimal wiring between rectangles*, Proceedings of the 13th STOC, Milwaukee, May, 1981, pp. 312-317.

[FR79] FAIRBAIRN, D.G. and J.A. ROWSON, ICARUS2 *user's manual*, Xerox Palo Alto Research Center Report, April 1980.

[F82] FIELD, D.E., *Drawing anti-aliased lines*, submitted for publication.

[FN81] NEWELL, M. and D. FITZPATRICK, *Magic - Multiple analyses of the geometry of integrated circuits*, Xerox Palo Alto Research Center Report, May 1981.

[KR82] KALIN, R. and N. RAMSEY, *Private communication.*

[L80] LaPAUGH, A.S., *A polynomial time algorithm for optimal routing around a rectangle*, Proceedings of 21st FOCS, Syracuse, New York, October 1980, pp. 282-293.

[LSV82] LIPTON, R.J., R. SEDGEWICK and J. VALDES, VLSI *layout as programming*, Proceedings of the Ninth POPL, Albuquerque, New Mexico,

January, 1982.

[MC80] MEAD, C. and L. CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[SH75] SHAMOS, M.I. and D. HOEY, *Closest point problems*, Proceedings of 16th FOCS, Berkeley, California, October, 1975, pp. 151-162.

[S80] STONE, H.S., *Introduction to Computer Architecture*, SRA, Palo Alto, 1980.

[T80] TOMPA, M., *An optimal solution to a wire routing problem*, Proceedings of 12th STOC, Los Angeles, California, May, 1980, pp. 161-176.

[V82] VALDES, J., *ALI2 System overview*, unpublished notes.

TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

(An asterisk before the MCT number indicates that the tract is under preparation).

A leaflet containing an order form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Orders should be sent to the same address.

MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963. ISBN 90 6196 002 9.

MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.

MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.

MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 005 3.

MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.

MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.

MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964. ISBN 90 6196 007 X.

MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964. ISBN 90 6196 008 8.

MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.

MCT 10 E.M. DE JAGER, *Applications of distrubutions in mathematical physics*, 1964. ISBN 90 6196 010 X.

MCT 11 A.R. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964. ISBN 90 6196 011 8.

MCT 12 J.A.Th.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965. ISBN 90 6196 013 4.

MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54.

MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.

MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.

MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967. ISBN 90 6196 022 3.

MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968. ISBN 90 6196 025 8.

MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968. ISBN 90 6196 038 X.

MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968. ISBN 90 6196 026 6.

MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.

MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968. ISBN 90 6196 028 2.

MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968. ISBN 90 6196 029 0.

MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.

MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.

MCT 25 E.R. PAËRL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.

MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968. ISBN 90 6196 031 2.

MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969. ISBN 90 6196 040 1.

MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969. ISBN 90 6196 041 X.

MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.

MCT 30 H. BRANDT CORSTIUS, *Exercises in computational linguistics*, 1970. ISBN 90 6196 052 5.

MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.

MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.

MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.

MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.

MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.

MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.

MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DUIJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium* 1971. ISBN 90 6196 065 7.

MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.

MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.

MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.

MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.

MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.

MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.

MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.

MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws* 1973. ISBN 90 6196 083 5.

MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.

MCT 47 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 2: The compiler*, 1973. ISBN 90 6196 085 1.

MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *An ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.

MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.

MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (eds), *Revised report on the algorithmic language ALGOL 68*, 1976. ISBN 90 6196 089 4.

MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.

MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.

MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.

MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.

MCT 55 M. HALL JR. & J.H. VAN LINT (eds), *Combinatorics, part 1: Theory of designs, finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.

MCT 56 M. HALL JR. & J.H. VAN LINT (eds), *Combinatorics, part 2: Graph theory, foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.

MCT 57 M. HALL JR. & J.H. VAN LINT (eds), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.

MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.

MCT 60 F. GÖBEL, *Queueing models involving buffers*, 1975. ISBN 90 6196 108 4.

*MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1*, ISBN 90 6196 109 2.

*MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2*, ISBN 90 6196 110 6.

MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.

MCT 64 W.J. DE SCHIPPER, *Symmetric closed categories*, 1975. ISBN 90 6196 112 2.

MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.

MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*, 1976. ISBN 90 6196 114 9.

*MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2*, ISBN 90 6196 119 X.

MCT 68 P.P.N. DE GROEN, *Singularly perturbed differential operators of second order*, 1976. ISBN 90 6196 120 3.

MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods*, 1977. ISBN 90 6196 125 4.

MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: Semantics and proof theory*, 1976. ISBN 90 6196 127 0.

MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes*, 1976. ISBN 90 6196 129 7.

MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of conical waveguides*, 1977. ISBN 90 6196 130 0.

MCT 73 D.M.R. LEIVANT, *Absoluteness of intuitionistic logic*, 1979. ISBN 90 6196 122 X.

MCT 74 H.J.J. TE RIELE, *A theoretical and computational study of generalized aliquot sequences*, 1976. ISBN 90 6196 131 9.

MCT 75 A.E. BROUWER, *Treelike spaces and related connected topological spaces*, 1977. ISBN 90 6196 132 7.

MCT 76 M. REM, *Associations and the closure statement*, 1976. ISBN 90 6196 135 1.

MCT 77 W.C.M. KALLENBERG, *Asymptotic optimality of likelihood ratio tests in exponential families*, 1977. ISBN 90 6196 134 3.

MCT 78 E. DE JONGE & A.C.M. VAN ROOIJ, *Introduction to Riesz spaces*, 1977. ISBN 90 6196 133 5.

MCT 79 M.C.A. VAN ZUIJLEN, *Empirical distributions and rank statistics*, 1977. ISBN 90 6196 145 9.

MCT 80 P.W. HEMKER, *A numerical study of stiff two-point boundary problems*, 1977. ISBN 90 6196 146 7.

MCT 81 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part 1, 1976. ISBN 90 6196 140 8.

MCT 82 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part 2, 1976. ISBN 90 6196 141 6.

MCT 83 L.S. BENTHEM JUTTING, *Checking Landau's "Grundlagen" in the AUTOMATH system*, 1979. ISBN 90 6196 147 5.

MCT 84 H.L.L. BUSARD, *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?) books vii-xii*, 1977. ISBN 90 6196 148 3.

MCT 85 J. VAN MILL, *Supercompactness and Wallman spaces*, 1977. ISBN 90 6196 151 3.

MCT 86 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix I, A programming system for operations on vectors and matrices over arbitrary fields and of variable size*. 1978. ISBN 90 6196 152 1.

*MCT 87 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix II*, ISBN 90 6196 153 X.

MCT 88 A. SCHRIJVER, *Matroids and linking systems*, 1977. ISBN 90 6196 154 8.

MCT 89 J.W. DE ROEVER, *Complex Fourier transformation and analytic functionals with unbounded carriers*, 1978. ISBN 90 6196 155 6.

MCT 90 L.P.J. GROENEWEGEN, *Characterization of optimal strategies in dynamic games*, 1981 . ISBN 90 6196 156 4.

MCT 91 J.M. GEYSEL, *Transcendence in fields of positive characteristic*, 1979. ISBN 90 6196 157 2.

MCT 92 P.J. WEEDA, *Finite generalized Markov programming*, 1979. ISBN 90 6196 158 0.

MCT 93 H.C. TIJMS & J. WESSELS (eds), *Markov decision theory*, 1977. ISBN 90 6196 160 2.

MCT 94 A. BIJLSMA, *Simultaneous approximations in transcendental number theory*, 1978. ISBN 90 6196 162 9.

MCT 95 K.M. VAN HEE, *Bayesian control of Markov chains*, 1978. ISBN 90 6196 163 7.

MCT 96 P.M.B. VITÁNYI, *Lindenmayer systems: Structure, languages, and growth functions*, 1980. ISBN 90 6196 164 5.

*MCT 97 A. FEDERGRUEN, *Markovian control problems; functional equations and algorithms*, . ISBN 90 6196 165 3.

MCT 98 R. GEEL, *Singular perturbations of hyperbolic type*, 1978. ISBN 90 6196 166 1.

MCT 99 J.K. LENSTRA, A.H.G. RINNOOY KAN & P. VAN EMDE BOAS, *Interfaces between computer science and operations research*, 1978. ISBN 90 6196 170 X.

MCT 100 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1*, 1979. ISBN 90 6196 168 8.

MCT 101 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2*, 1979. ISBN 90 6196 169 6.

MCT 102 D. VAN DULST, *Reflexive and superreflexive Banach spaces*, 1978. ISBN 90 6196 171 8.

MCT 103 K. VAN HARN, *Classifying infinitely divisible distributions by functional equations*, 1978. ISBN 90 6196 172 6.

MCT 104 J.M. VAN WOUWE, *Go-spaces and generalizations of metrizability*, 1979. ISBN 90 6196 173 4.

MCT 105 R. HELMERS, *Edgeworth expansions for linear combinations of order statistics*, 1982. ISBN 90 6196 174 2.

MCT 106 A. SCHRIJVER (ed.), *Packing and covering in combinatorics*, 1979. ISBN 90 6196 180 7.

MCT 107 C. DEN HEIJER, *The numerical solution of nonlinear operator equations by imbedding methods*, 1979. ISBN 90 6196 175 0.

MCT 108 J.W. DE BAKKER & J. VAN LEEUWEN (eds), *Foundations of computer science III*, part *1*, 1979. ISBN 90 6196 176 9.

MCT 109 J.W. DE BAKKER & J. VAN LEEUWEN (eds), *Foundations of computer science III*, part *2*, 1979. ISBN 90 6196 177 7.

MCT 110 J.C. VAN VLIET, *ALGOL 68 transput*, part *I*: *Historical review and discussion of the implementation model*, 1979. ISBN 90 6196 178 5.

MCT 111 J.C. VAN VLIET, *ALGOL 68 transput*, part *II*: *An implementation model*, 1979. ISBN 90 6196 179 3.

MCT 112 H.C.P. BERBEE, *Random walks with stationary increments and renewal theory*, 1979. ISBN 90 6196 182 3.

MCT 113 T.A.B. SNIJDERS, *Asymptotic optimality theory for testing problems with restricted alternatives*, 1979. ISBN 90 6196 183 1.

MCT 114 A.J.E.M. JANSSEN, *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes*, 1979. ISBN 90 6196 184 X.

MCT 115 P.C. BAAYEN & J. VAN MILL (eds), *Topological Structures II*, part *1*, 1979. ISBN 90 6196 185 5.

MCT 116 P.C. BAAYEN & J. VAN MILL (eds), *Topological Structures II*, part *2*, 1979. ISBN 90 6196 186 6.

MCT 117 P.J.M. KALLENBERG, *Branching processes with continuous state space*, 1979. ISBN 90 6196 188 2.

MCT 118 P. GROENEBOOM, *Large deviations and asymptotic efficiencies*, 1980. ISBN 90 6196 190 4.

MCT 119 F. J. PETERS, *Sparse matrices and substructures, with a novel implementation of finite element algorithms*, 1980. ISBN 90 6196 192 0.

MCT 120 W.P.M. DE RUYTER, *On the asymptotic analysis of large-scale ocean circulation*, 1980. ISBN 90 6196 192 9.

MCT 121 W.H. HAEMERS, *Eigenvalue techniques in design and graph theory*, 1980. ISBN 90 6196 194 7.

MCT 122 J.C.P. BUS, *Numerical solution of systems of nonlinear equations*, 1980. ISBN 90 6196 195 5.

MCT 123 I. YUHÁSZ, *Cardinal functions in topology - ten years later*, 1980. ISBN 90 6196 196 3.

MCT 124 R.D. GILL, *Censoring and stochastic integrals*, 1980. ISBN 90 6196 197 1.

MCT 125 R. EISING, *2-D systems, an algebraic approach*, 1980. ISBN 90 6196 198 X.

MCT 126 G. VAN DER HOEK, *Reduction methods in nonlinear programming*, 1980. ISBN 90 6196 199 8.

MCT 127 J.W. KLOP, *Combinatory reduction systems*, 1980. ISBN 90 6196 200 5.

MCT 128 A.J.J. TALMAN, *Variable dimension fixed point algorithms and triangulations*, 1980. ISBN 90 6196 201 3.

MCT 129 G. VAN DER LAAN, *Simplicial fixed point algorithms*, 1980. ISBN 90 6196 202 1.

MCT 130 P.J.W. TEN HAGEN et al., *ILP Intermediate language for pictures*, 1980. ISBN 90 6196 204 8.

MCT 131 R.J.R. BACK, *Correctness preserving program refinements: Proof theory and applications*, 1980. ISBN 90 6196 207 2.

MCT 132 H.M. MULDER, *The interval function of a graph*, 1980. ISBN 90 6196 208 0.

MCT 133 C.A.J. KLAASSEN, *Statistical performance of location estimators*, 1981. ISBN 90 6196 209 9.

MCT 134 J.C. VAN VLIET & H. WUPPER (eds), *Proceedings international conference on ALGOL 68*, 1981. ISBN 90 6196 210 2.

MCT 135 J.A.G. GROENENDIJK, T.M.V. JANSSEN & M.J.B. STOKHOF (eds), *Formal methods in the study of language*, part *I*, 1981. ISBN 90 6196 211 0.

MCT 136 J.A.G. GROENENDIJK, T.M.V. JANSSEN & M.J.B. STOKHOF (eds), *Formal methods in the study of language*, part *II*, 1981. ISBN 90 6196 213 7.

MCT 137 J. TELGEN, *Redundancy and linear programs*, 1981. ISBN 90 6196 215 3.

MCT 138 H.A. LAUWERIER, *Mathematical models of epidemics*, 1981. ISBN 90 6196 216 1.

MCT 139 J. VAN DER WAL, *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games*, 1980. ISBN 90 6196 218 8.

MCT 140 J.H. VAN GELDROP, *A mathematical theory of pure exchange economies without the no-critical-point hypothesis*, 1981.
ISBN 90 6196 219 6.

MCT 141 G.E. WELTERS, *Abel-Jacobi isogenies for certain types of Fano three-folds*, 1981.
ISBN 90 6196 227 7.

MCT 142 H.R. BENNETT & D.J. LUTZER (eds), *Topology and order structures*, part 1, 1981.
ISBN 90 6196 228 5.

MCT 143 H. J.M. SCHUMACHER, *Dynamic feedback in finite- and infinite dimensional linear systems*, 1981.
ISBN 90 6196 229 3.

MCT 144 P. EIJGENRAAM, *The solution of initial value problems using interval arithmetic. Formulation and analysis of an algorithm*, 1981.
ISBN 90 6196 230 7.

MCT 145 A.J. BRENTJES, *Multi-dimensional continued fraction algorithms*, 1981. ISBN 90 6196 231 5.

MCT 146 C. VAN DER MEE, *Semigroup and factorization methods in transport theory*, 1982. ISBN 90 6196 233 1.

MCT 147 H.H. TIGELAAR, *Identification and informative sample size*, 1982.
ISBN 90 6196 235 8.

MCT 148 L.C.M. KALLENBERG, *Linear programming and finite Markovian control problems*, 1983. ISBN 90 6196 236 6.

MCT 149 C.B. HUIJSMANS, M.A. KAASHOEK, W.A.J. LUXEMBURG & W.K. VIETSCH, (eds), *From A to Z, proceeding of a symposium in honour of A.C. Zaanen*, 1982. ISBN 90 6196 241 2.

MCT 150 M. VELDHORST, *An analysis of sparse matrix storage schemes*, 1982.
ISBN 90 6196 242 0.

MCT 151 R.J.M.M. DOES, *Higher order asymptotics for simple linear Rank statistics*, 1982. ISBN 90 6196 243 9.

MCT 152 G.F. VAN DER HOEVEN, *Projections of Lawless sequences*, 1982.
ISBN 90 6196 244 7.

MCT 153 J.P.C. BLANC, *Application of the theory of boundary value problems in the analysis of a queueing model with paired services*, 1982.
ISBN 90 6196 247 1.

MCT 154 H.W. LENSTRA, JR. & R. TIJDEMAN (eds), *Computational methods in number theory, part I*, 1982.
ISBN 90 6196 248 X.

MCT 155 H.W. LENSTRA, JR. & R. TIJDEMAN (eds), *Computational methods in number theory, part II*, 1982.
ISBN 90 6196 249 8.

MCT 156 P.M.G. APERS, *Query processing and data allocation in distributed database systems*, 1983.
ISBN 90 6196 251 X.

MCT 157 H.A.W.M. KNEPPERS, *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic*, 1983.
ISBN 90 6196 252 8.

MCT 158 J.W. DE BAKKER & J. VAN LEEUWEN (eds), *Foundations of computer science IV, Distributed systems*, part 1, 1983.
ISBN 90 6196 254 4.

MCT 159 J.W. DE BAKKER & J. VAN LEEUWEN (eds), *Foundations of computer science IV, Distributed systems*, part 2, 1983.
ISBN 90 6196 255 0.

MCT 160 A. REZUS, *Abstract automat*, 1983.
ISBN 90 6196 256 0.

MCT 161 G.F. HELMINCK, *Eisenstein series on the metaplectic group, An algebraic approach;* 1983.
ISBN 90 6196 257 9.

An asterisk before the number means "to appear"