DISTRIBUTED COMPUTING

Jan van Leeuwen

RUU-CS-82-8

June 1982

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

DISTRIBUTED COMPUTING

Jan van Leeuwen

Technical Report RUU-CS-82-8

June 1982

Department of Computer Science
University of Utrecht
P.O. Box 80.002
3508 TA Utrecht
the Netherlands

DISTRIBUTED COMPUTING*

Jan van Leeuwen

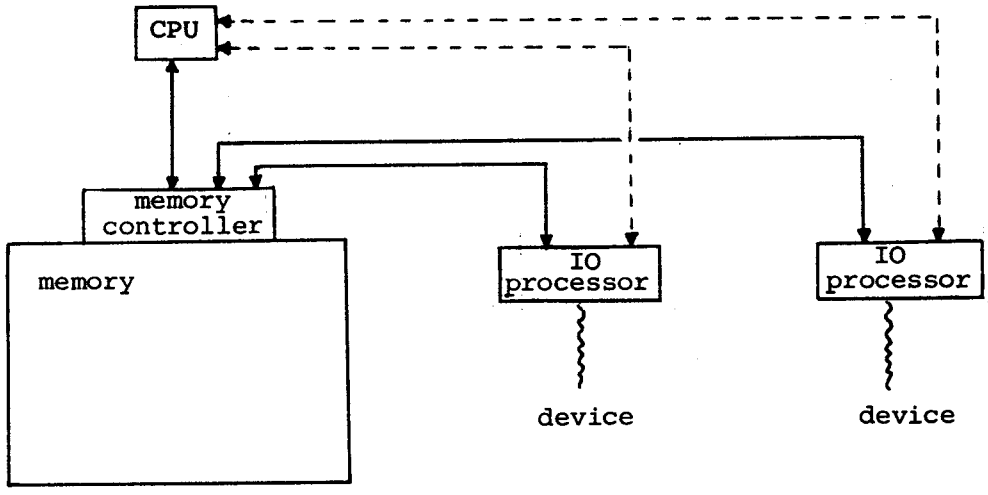
Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands.

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].

---

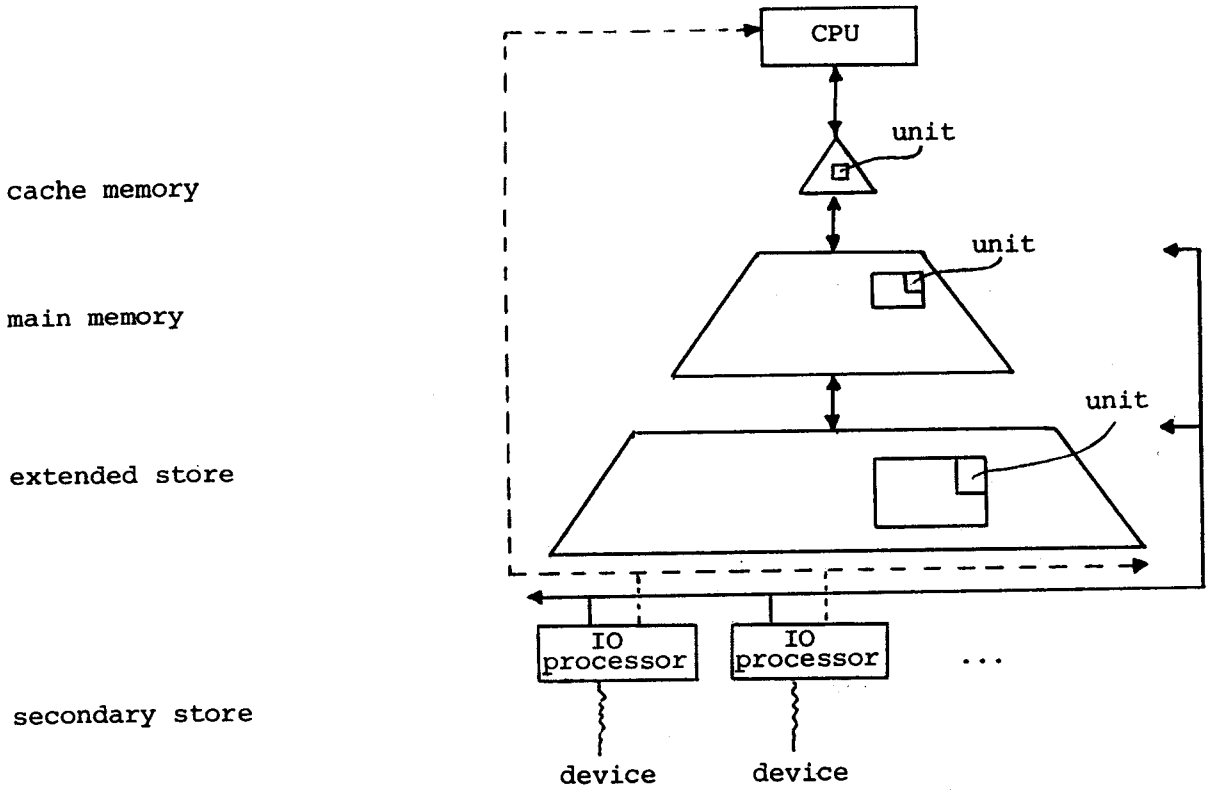
* Notes for the opening presentation of the 4th Advanced Course on Foundations of Computer Science, held June 14-25 (1982) in Amsterdam.

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 $u \leq v \Rightarrow f(u) \leq f(v)$) 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, $\geq$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.

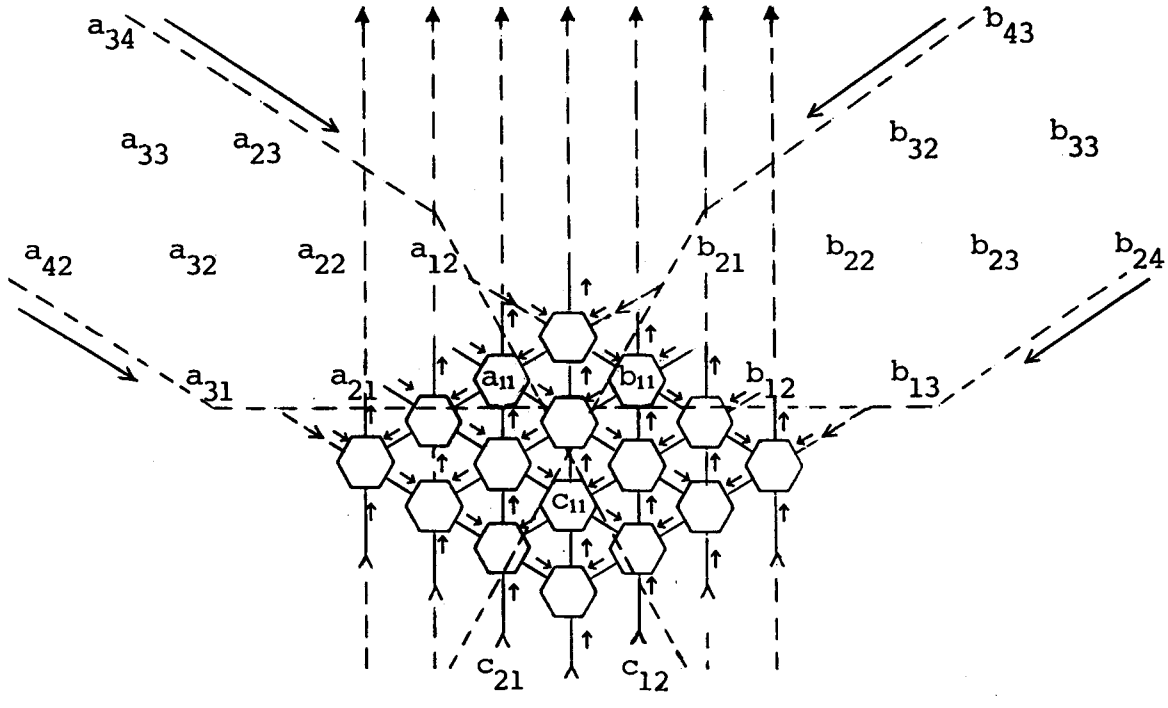
11. Systolic algorithms. Given the current technology it has become feasible to design chips for every imaginable special-purpose function in a system, an approach advocated by Kung [45] (see also [46]). The chip design must begin with a distributed algorithm design, conceptually specifying the overal 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.

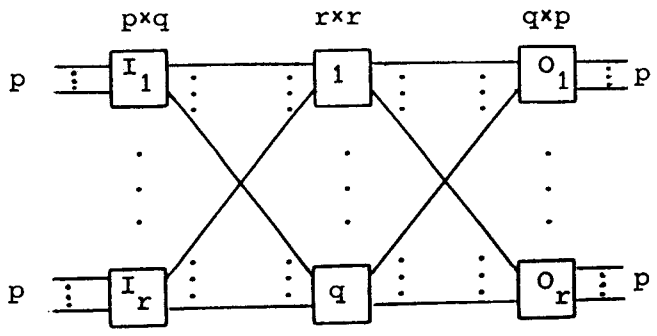
Proof

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 $\lfloor i_M \cdots i_0 \rfloor \overrightarrow{j_M \cdots j_0}$ 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 $\{K_{i_m}\}_{1\le m\le s}$. 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 $|\bigcup_1^s K_{i_m}| \ge s$, which is Hall's condition ([56]) for the existence of a set of distinct representatives. Let j_m be the representative of K_m ($1\le m\le r$) 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. □

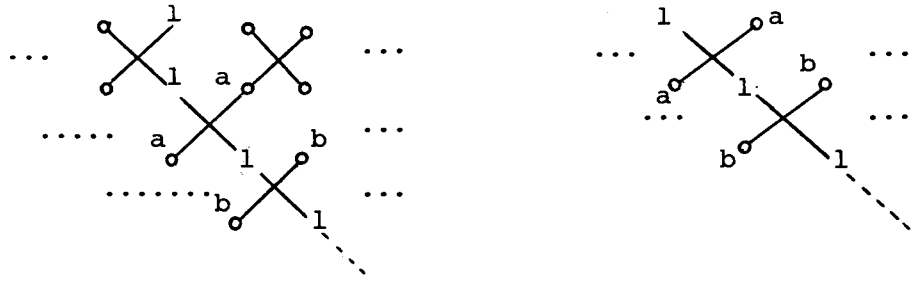
Note that we haven't used that each box is a crossbar, but merely that it is rearrangeable. This allows a recursive construction of rearrangeable networks. In particular, a Benes network B(n) is a Clos network $C(2,2,2^{n-1})$ with two B(n-1)'s as intermediate crossbars. The network has size O(N log 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

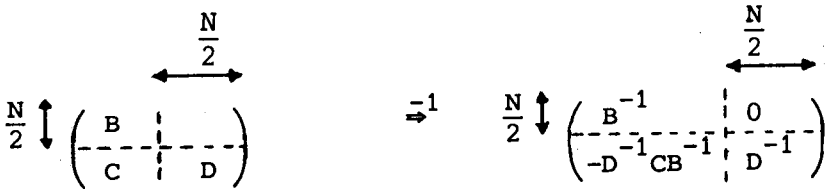
Use processors P_i ($2 \leq i \leq N$). After eliminating x_{j-1} ($j \geq 2$) assume that the P_i with $i \geq j$ have $a_{i1}x_1 + \ldots + a_{ij-1}x_{j-1}$ in store. In the next cycle P_j can compute x_j. It subsequently broadcasts the value to all P_i with i>j, which compute $a_{ij}.x_j$ and add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

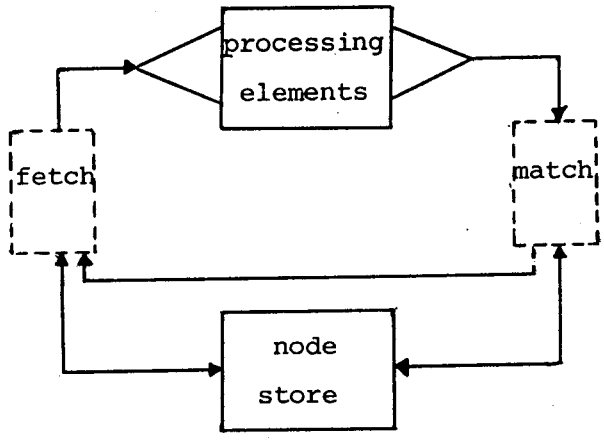
The timing of Sollin's algorithm is different depending on the memory model used (we allowed "unambiguous" access conflicts). Bentley and Ottmann [70] proved that the algorithm can run in O(N log N) time on a linear array of N processors.

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 different 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

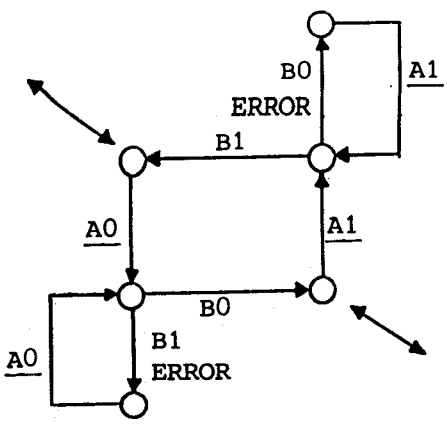
used to decide if a packet is to be received, and (iii) the packet erase policy (to use in case a packet appears to circulate indefinitely). There are three major types of ring architectures in use, which differ largely by the transmission policy that is used: (i) slotted rings, (ii) token rings and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

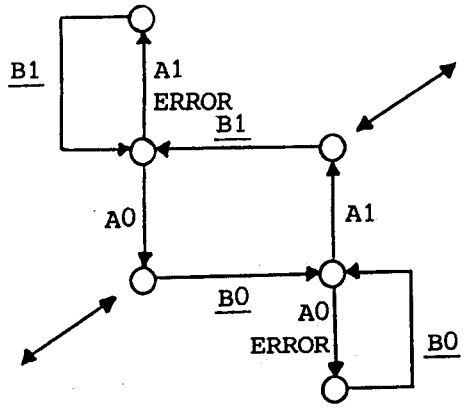
Theorem. One bit suffices for error control when transmitting over a half-duplex line.

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.

[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.