Non-Sequential Computation and Laws of Nature

P.M.B. Vitányi
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Traditionally, computational complexity theory deals with sequential computations. In the computational models the underlying physics is hardly accounted for. This attitude has persisted in common models for parallel computations. Wrongly, as we shall argue, since the laws of physics intrude forcefully when we want to obtain realistic estimates of the performance of parallel or distributed algorithms. First, we shall explain why it is reasonable to abstract away from the physical details in sequential computations. Second, we show why certain common approaches in the theory of parallel complexity do not give useful information about the actual complexity of the parallel computation. Third, we give some examples of the interplay between physical considerations and actual complexity of distributed computations.

1980 Mathematics Subject Classification: 68C05, 68C25, 68A05, 68B20, 94C99
CR Categories: B.7.0, C.2, D.4, F.2.2, F.2.3, G.2.2.
Keywords and Phrases: sequential computation, parallel computation, distributed computation, VLSI, computational complexity, time, space, physics, communication, wires, limitations, laws of nature
Note: Invited lecture at the *Aegean Workshop on Computing, VLSI Algorithms and Architectures* (2nd international Workshop on parallel computing and VLSI), Loutraki, Greece, July 8-11, 1986.

1. Introduction

The earliest electronic computing engines arose as a byproduct of the Manhattan Project in World War II. Broadly speaking, their purpose was to compute numerical solutions to second order partial differential equations arising in connection with the design of the atomic bomb. The machines consisted of primitive logical and memory components like electromagnetic relays and mercury delay lines, which where wired up so as to have the complex perform the desired computation. The *architecture* reflected the type of *algorithm* to

This work was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grant DCR-83-02091, and by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-83-K-0125.

Report CS-R8618
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

be performed, i.c., the solution of the mentioned equations by numerical grid methods. Such algorithms suggest parallel or pipelined execution, and that is exactly the type of architecture of those first computers [3]. Only at the present time, in the middle eighties, have we come full circle and see such special purpose architectures again in the pipelined and systolic algorithms frozen in the silicon hardware of chips. Once more, the shift is away from sequential thinking in the form of line-by-line programs of imperative or other nature, and to representing algorithms in structures of space and time.

After the Manhattan Project had been fulfilled, computer designers quickly progressed to the idea of automating all types of computational tasks. Rather than stooping to the chore of rewiring a new complex for every new task which came along, the idea arose of letting the computer take over that job as well. Thus, the idea of a *general purpose* computer entered the scene. It so happened that mathematicians like H.H. Goldstine, J. von Neumann and A.W. Burks were well aware of A.M. Turing's brilliant 1936 paper [9] in which he described an architecture for just such a hypothetical machine:

"Computing is normally done by writing certain symbols on paper. We may suppose this paper to be divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares. I also suppose that the number of symbols which may be printed is finite."

"The behaviour of the [human] computer at any moment is determined by the symbols he is observing, and his 'state of mind' at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite."

"We suppose [above] that the computation is carried out on a tape; but we avoid introducing the "state of mind" by considering a more physical and definitive counterpart of it. It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. This note is the counterpart of "the state of mind." We will suppose that the computer works in such a desultory manner that he never does more than one step and write the next note. Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. That is, the state of the system may be described by a single expression (sequence of symbols), consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression may be called the "state formula." We know that the state formula at any given stage is determined by the state formula before the last step was made, and we assume that the relation of these two formulae is expressible in the functional calculus. In other words, we assume that there is an axiom A which expresses the rules governing the behaviour of the computer, in terms of the relation of the state formula at any stage to the state formula at the preceding stage. If this is so, we can construct a machine to write down the successive state formulae, and hence to compute the required number."

Grasping the implied architectural concept, and improving it according to the leeway provided by physical law, Burks, Goldstine and von Neumann in 1946 wrote a memorandum [1] which shaped the architecture of electronic computers for the next forty years. This memorandum was preceded by the famous 'First Draft' [8], where we can clearly distinguish the serial mode of operation of the modern computer, i.e., one instruction at a time is inspected and then executed. This is in sharp distinction to the parallel operation of the earlier ENIAC computer in which many things where simultaneously being performed. To

abandon all parallelism was not thought of as detrimental to performance, since the potential speed of the electronic techniques was judged to be fast enough. Complainants about the 'von Neumann' bottleneck (explained below), inherent in the stored program sequential computer as we know it, should realize that the conceptual advantage of this scheme is what made possible the giant strides of progress: if cars had become so much cheaper as computing power has, a car would cost less than 1 dollar.

Turing's analysis of the process of computation as the sequential execution of a sequence of operations is so natural, that it seems as if Euclid in designing one of the earliest known algorithms (for computing the greatest common divisor) must have had such an architecture in mind. Now it so happens, that in sequential computation we can ignore many physical details of the underlying computer system in analysing the computational complexity of some program. Each operation essentially consists of a sequence of "fetch from memory," "execute operation on one or more operands in the Central Processing Unit" and "store in memory." The CPU operations can be thought of - when viewed from sufficient distance - as essentially finite automata transitions which transform input obtained by a bounded number of "fetch from memory" operations (say 2) into output in the form of "store in memory" operations (say 1). In the usual setup, a memory register has a fixed length (say 48 bits) and both the memory accesses and CPU operations take a fixed time (say, at most X). Therefore, a sequence of n operations takes in between nX and $4nX$ time. Forgetting about the X and the small constants like 4, it is usual to say that n operations take n 'time.' Note, here 'time' means number of steps. Similarly, it is assumed that all objects manipulated fit in a single memory location. Moreover, that each object is 'random accessible,' that is, each object can be accessed as fast as any other. This is referred to as the 'unit cost measure.'

This scheme is sometimes refined to take into account that some items being manipulated do not fit in a 48 bit register - as for instance the 123rd Mersenne prime. It is then customary to charge the cost of manipulating the item as being linear in its length, both in terms of storage and in terms of time for execution of an operation. This is referred to as the 'logarithmic cost measure.' It is clear, that this time cost measure is only a lower bound, since the actual operations performed on the items when they are chopped up, often requires more than time linear in the length of the items. For instance, while logarithmic cost may be reasonable for addition, it is not reasonable for multiplication.

A further refinement may be made for objects not held in 'random access' memory, but on disk or mass storage devices such as tapes. There an operation on an object may involve swapping pieces of the object back and forth from disk to random access memory, thus incurring a time overhead which may be orders of magnitudes greater than the time spent on manipulating in the CPU and random access memory. Think about the sorting or merging of huge data files. The logarithmic cost measure tries to take such an overhead into account by charging as the cost of a memory access also the length of the memory address. As in the case of the registers, this can be only a very crude lower bound on the actual cost. We thus distinguish a memory hierarchy, where the access times of objects stored at different levels differs orders of magnitudes.

While the physical aspects of computing devices can thus be fairly well accounted for, the basic unit of time a transaction takes does not vary too wildly within each level we have distinguished. It is therefore more or less justified to forget about the details and talk only about the number of operations at each level of the memory hierarchy. As we will see, in

the realm of nonsequential computation reality can not be ignored to such an extent.

Since in current computers the time of a basic operation in the CPU is generally far lower than that of memory accesses, most computations are memory bound, i.e., the time spent in accessing various levels in the memory hierarchy completely dominates the computation time. This is popularly called the 'von Neumann' bottleneck. Are the prospects any brighter in the coming era of nonsequential computation?

2. Space

In many areas of the theory of parallel computation we meet tree structured devices or computations.

(1). For instance, 'parallel random access machines (PRAM's)' can at each point in their computation spawn a couple of offspring PRAM's to perform some subcomputations. Broadly speaking, we can therefore imagine the computation as a binary tree of processors. The 'time' the computation takes is then linearly related to the depth of the tree.

(2). In [5] this idea is translated into terms of 'very large integrated circuits.' In Chapter 8 the authors show a bold picture of a complete binary tree, and explain that such a tree with processors in each node, is capable of solving NP-complete problems like the 'traveling salesman problem' in linear time. This, on the grounds that the processor at the root can send a copy of the problem instance to each of the leaves, and each of the leaves can try one candidate solution. A simple scheme can guarantee that each leaf tries a different solution, each solution is tried by some leaf, and all answers are percolated upwards to the root. If positive answers win over negative ones in the fan in, the answer the root receives is a solution if there is one and 'no solution' if there is none.

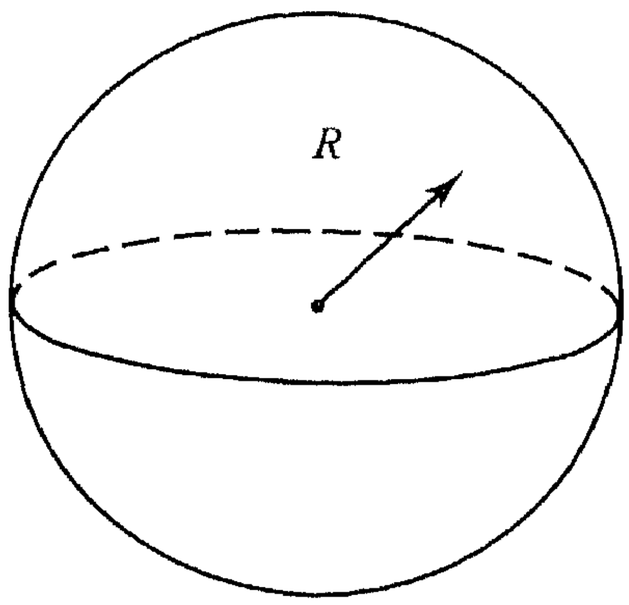
(3). One of the currently flourishing parts of the theory of parallel computation is 'NC-computation.' A problem is in 'Nick's Class' if it can be solved in polylogarithmic 'time' using a polynomial number of processors. Here, 'time' means the length of the longest chain of causally related steps.

All of the above models may say something about the parallelizability of algorithms for certain problems. This often takes the form of distributing copies of the entire problem instance, or pieces of the problem instance, among an exponential number of processors in a linear number of steps. Or, as in NC, among a polynomial number of processors in a polylogarithmic number of steps. The way a problem instance can be divided and partial answers put together may give genuine insight into its parallelizability. However, it can *not* give a reduction from an asymptotic exponential time best algorithm in the sequential case to an asymptotic polynomial time algorithm in *any* parallel case. At least, if by 'time' we mean time. This can be seen easily as follows. If the parallel algorithm uses 2^n processing elements, regardless of whether the computational model assumes bounded fan-in and fan-out or not, it can not run in time polynomial in n, because *physical space* has us in its tyranny. Viz., if we use 2^n processing elements of, say, unit size each, then the tightest they can be packed is in a 3-dimensional sphere of volume 2^n. No unit in the sphere can be closer to all other units than a distance of radius R,

$$R = \left[\frac{3 \cdot 2^n}{4\pi}\right]^{1/3}$$

Modulo a major advance in physics, it is impossible to transport signals over $2^{\alpha n}$ ($\alpha>0$) listance in polynomial $p(n)$ time. In fact, the assumption of the bounded speed of light uggests that the lower time bound on *any* computation using 2^n processing elements is $\Omega(2^{n/3})$ outright. Or, for the case of NC computations which use n^α processors, $\alpha>0$, the ower bound on the computation time is $\Omega(n^{\alpha/3})$.

The situation is worse than it appears on the face of it. Consider an architecture such as he binary n-cube. This is the network in which the nodes are identified by n-bit names, ınd there is a communication edge between two nodes if their identifiers differ in a single ›it. Call this graph $C=(V,E)$. Let C be embedded in 3-dimensional Euclidean space, and et each node have unit volume. Let x be any node of C. There are at most $2^n/8$ nodes vithin Euclidean distance $R/2$ of x, where R is as above. Then, there are $\geqslant 7\cdot 2^n/8$ nodes at Euclidean distance $\geqslant R/2$ from x.



Construct a spanning tree T_x of in C of depth $\leqslant n$ with node x as the root. The average Euclidean length of a path from the root in T_x is $\geqslant 7R/16$, and therefore the average Euclidean length of an embedded edge in a path from the root in T_x is $\geqslant 7R/16n$. This does not give a lower bound on the average Euclidean length of an edge in T_x. However, using the symmetry of the binary n-cube we can establish that the average Euclidean length of the edges in the 3-space embedding of C is $\geqslant 7R/16n$. We can prove this as follows. (The hasty reader may skip the proof by proceeding to the second column on the next page.)

Proof. Denote a node a in C by a n-bit string $a_1a_2\cdots a_n$, and an edge (a,b) between nodes a and b differing in the ith bit by:

$$a = a_1\cdots a_{i-1}\hat{a}_i a_{i+1}\cdots a_n$$

This means that an edge has two representations. Now we can express a set I of isomorphic mappings of C to itself by (1) a cyclic permutation of the representation of nodes and edges, followed by (2) complementation of the bits of the representations in a given pattern. I.e., the isomorphism $(j,c_1c_2\cdots c_n)\in I$ maps the above edge a to

$$b = b_{j+1}\cdots b_{i-1}\hat{b}_i b_{i+1}\cdots b_n b_1\cdots b_j$$

with $b_i=a_i$ if $c_i=0$ and $b_i=\overline{a_i}$ ($=$ complement a_i) if $c_i=1$.

Consider the ensemble S of spanning trees of C, each tree isomorphic with T_x above, consisting of the $n2^n$ trees $i(T_x)$ to which T_x is mapped by the $n2^n$ distinct isomorphisms i in I. For each edge e in T_x and each edge e' of C there are two distinct isomorphisms i_1 and i_2 in I such that $i_1(e)=i_2(e)=e'$. The average Euclidean length of a path from the root in each tree $i(T_x)\in S$ $(i\in I)$ is $\geqslant 7R/16$, so the average Euclidean length of a path from the root taken over all trees $i(T_x)\in S$ $(i\in I)$ is $\geqslant 7R/16$ as well. Let the Euclidean length of an edge e in the 3-space embedding of C be $l(e)$. Then, for each edge e of T_x:

$$\sum_{i\in I} l(i(e)) = 2\sum_{e\in E} l(e)$$

That is, each edge in the embedded C occurs twice as the same edge of the canonical tree T_x in the form of the corresponding isomorphic edge in some tree in S. Therefore, the average Euclidean length of the edges in trees in S, which correspond to a single particular edge of T_x, equals the average Euclidean length of an edge in E. Let P be a path from the root in T_x consisting of $|P|\leqslant n$ edges. Then, the average sum of the Euclidean lengths of the edges in a path $i(P)$ from the root in all trees $i(T_x)$ $(i\in I)$ equals $|P|$ times the average Euclidean edge length in E:

$$\sum_{e\in P}\sum_{i\in I} l(i(e)) = 2|P|\sum_{e\in E} l(e)$$

Consequently, the average Euclidean edge length in E equals the average Euclidean length of an edge in a path P from the root in a tree in S, and is therefore $\geqslant 7R/16n$:

$$\frac{\sum_{e\in E} l(e)}{n2^{n-1}} = \frac{\sum_{P\in T_x} \frac{\sum_{e\in P}\sum_{i\in I} l(i(e))}{n2^n|P|}}{2^n}$$

$$\geqslant \frac{7R}{16n}$$

Since there are $n2^n/2$ edges in the binary n-cube, this sums up to an amazing *total* wire length $\sum_{e\in E} l(e)$ needed in the Euclidean 3-dimensional embedding of C of

$$\sum_{e\in E} l(e) \geqslant \frac{2^n 7R}{32}$$

$$\geqslant \left(\frac{3}{4\pi}\right)^{1/3} \cdot 7 \cdot 2^{(4n/3)-5}$$

Many network topologies are afflicted with this problem: n-dimensional cube networks, fast Fourier networks, butterfly networks, shuffle-exchange networks, cube-connected cycles networks, and so on. In fact, the arguments seem to hold for networks with a small diameter which satisfy certain symmetry requirements. An example of a network with small diameter which is not symmetric in this sense is the tree. The fact that 7/8th of all paths from the root in a complete tree would have Euclidean length $\geqslant R/2$ in a 3-space embedding do not imply that the average Euclidean length of an embedded edge of the tree is larger than a constant. This is borne out by the familiar H-tree layout [5] where the average edge length is less than 3 or 4. However, in the recently investigated 'fat tree' architectures the wire length will dominate again. In a complete binary fat tree of depth n and root at level

0, a node at level $i+1$ is connected to a node at level i by a 'bundle' of 2^{n-i} edges. Then, trivially, the average Euclidean length of an edge in a path from the root equals the average Euclidean length of an edge in the fat tree, leading to the result above.

Note. Deriving the result about the total necessary wire length for embedding the binary n-cube, we did not make *any* assumptions about the volume of a wire of unit length, or the way they are embedded in space, as is usual [10]. It is consistent with the derived results that wires have *zero* volume, and that *infinitely* many wires can pass through a unit 2-dimensional area. Such assumptions invalidate the arguments used elsewhere. In contrast with other investigations, the goal here is to derive lower bounds on the total wire length irrespective of the ratio between the volume of a unit length wire and the volume of a processing element. The lower bound on the total wire length above is independent of this ratio, which changes with different technologies or granularity of computing components.

Iterating the above reasoning, but now adding the volume of the wires to the volume of the nodes, the greatest lower bound on the volume necessary to embed the binary n-cube converges to a particular solution in between a total volume of $\Omega(2^{4n/3})$ and a total volume of, say, $O(2^{2n})$ if we charge a constant fraction of the unit volume for a unit wire length. The lower bound $\Omega(2^{4n/3})$ ignores the fact that the added volume of the wires pushes the nodes further apart, thus necessitating longer wires again. The $O(2^{2n})$ upper bound holds under the assumption that wires of all lengths have the same volume per unit length (not more than a constant fraction of the unit volume of a node). In a later section I show that the latter assumption cannot always be made.

These surprising facts are a theoretical prelude to many *wiring problems* currently starting to plague computer designers and chip designers alike. Formerly, a wire had magical properties of transmitting data 'instantly' from one place to another (or better, to many other places). A wire did not take room, did not dissipate heat, and did not cost anything - at least, not enough to worry about. This was the situation when the number of wires was low, somewhere in the hundreds. Current designs use many millions of wires (on chip), or possibly billions of wires (on wafers). In a computation of parallel nature, most of the time seems to be spent on communication - transporting signals over wires. Thus, thinking that the von Neumann bottleneck has been conquered by nonsequential computation, we are unaware that the Non-von Neumann bottleneck is still waiting. The following innominate quote covers this matter admirably:

> "Without me they fly they think; But when they fly I am the wings."

Another effect which becomes increasingly important is that most of the room in the device executing the computation is taken up by the wires. Under very conservative estimates that the unit length of a wire has a volume which is a constant fraction of that of a component it connects, we can see above that in 3-dimensional layouts for binary n-cubes, or for the other fast permutation networks, the volume of the 2^n components performing the actual computation operations is an asymptotic fastly vanishing fraction of the volume of the wires needed for communication:

$$\frac{\text{volume computing components}}{\text{volume communication wires}} \in o(2^{-n/3})$$

Today it seems that a partial solution to this problem can be found in optical communication, either wireless by means of lasers/infrared light or by using virtually unlimited bandwidth glass fiber. But beware, even while Nature is not malicious, she is subtle.

3. Time

It is useful to distinguish between *distributed computation* and *distributed control*. Whereas the former is concerned with the distributed solution of problems for which there also exist sequential algorithms, the latter is concerned with problems which make no sense in terms of sequential computation. Examples of the former are parallel algorithms for matrix multiplication, fast Fourier transform, shortest path, matching. Examples of the latter are methods for mutual exclusion and nameserver [7], distributed spanning tree, clock synchronization algorithms, Byzantine agreement, leader election, symmetry breaking. In distributed control the notion of *time* plays an all-important role.

As large multiprocessor systems communicating by message passing start to be actually constructed, and on a geographically grander scale very large computer networks, synchronization problems connected with the operation of such complexes are bound to become acute. Another problem which gets crucial for very large computer complexes is the number of message passes. Without efficient congestion control the system will be swamped by communication messages effectively blocking throughput.

To fix thoughts, the networks we consider are point-to-point (store-and-forward) communication networks described by an undirected communication graph, with the set of nodes representing the processors of the network, and the set of links representing bidirectional noninterfering communication channels between them. No common memory is shared by the node-processors. Each node processes messages received from its neighbors, performs local computations on messages and sends messages to neighbors. All these actions take a finite time. All messages have a finite length according to the finite amount of information they carry. Each message sent by a node to its neighbor arrives there in a finite time. A *message pass* consists of the sending of a message from one node to one of its direct neighbors. In order to make the cost measure meaningful, when we express the complexity of some algorithm in the number of message passes, we want to exclude unrealistically long messages. One choice is to allow messages of size $O(\log n)$, where n is the number of nodes in the network. The *time* complexity of a distributed algorithm should obviously be the size of the interval between the beginning and the end of the algorithm. As yet there seems to be no completely satisfactory general method to compute this cost constructively, given the algorithm, for the many types of distributed algorithms which are known. However, this is only one of many problems associated with the concept of time in distributed systems.

Here we focus on problems resulting from lack of synchronization. These can be dealt with using 'partially ordered' time, as in [4], or by constructing algorithms that can deal with unlimited asynchrony. The latter algorithms can surely deal with any environment in which there is knowledge about processor speed and message delivery time. Unlimited asynchronous models have been thoroughly investigated, as have purely synchronous models. Physical systems are usually somewhere in between: they are neither purely synchronous nor unlimited asynchronous. It is therefore an interesting exercise to develop algorithms that do not use knowledge about the relative progress of time in the system, yet perform superior under *realistic* conditions about time. The usual logically time-independent algorithms do not assume anything about the rate at which time flows in different locations. This is unnecessarily harsh with respect to many problems arising in the real world. Clock drift in systems happens with a certain smoothness, since abrupt changes are rare in nature. It seems to be worthwhile to investigate robust algorithms such that:

- the algorithms remain correct and terminate under any behavior of time in the system,
- using time, the algorithms are yet logically time-independent, only their efficiency depends on the behavior of time,
- with increasing synchronous well-behaved time in the system the performance of the algorithm improves ever faster,
- if the asynchrony of the system is known then the algorithm performs as well as in the synchronous case,
- under practical assumptions about clock speeds these algorithms use less message passes than is possible by any other known methods for the problems they solve in asynchronous systems,
- the limitation on unlimited asynchrony such algorithms require is but a minor one which is generally satisfied and which we term "Archimedean asynchronicity".

Now, in asynchronous distributed systems each processor has its own clock. Although these clocks may not be synchronized, and the clocks may not indicate the same time, there should be some proportion between the clock rates. That is, if an interval of time has passed on the clock for processor A, a *proportional* period of time has passed on the clock for processor B.

Definition. A distributed system is *Archimedean* from time t_1 to time t_2 if the ratio of the time intervals between the ticks of the clocks of any pair of processors, and the ratio between the communication delay between any adjacent pair of processors and the time interval between the ticks of the clock of any processor, is bounded by a fixed integer during the time interval from t_1 to t_2. (This ratio need not be bounded a priori, nor need it be known to the processors concerned.)

That is, in asynchronous networks the magnitudes of elapsed time should satisfy the axiom of Archimedes. The axiom of Archimedes holds for a set of magnitudes if, for any pair a, b of such magnitudes, there is a multiple na which exceeds b for some natural number n. It is called *Archimedes' axiom** possibly due to an application in obtaining large numbers in *The Sand-Reckoner.*

We assume that the magnitudes of elapsed time, as measured, for instance, by local clocks amongst different processors or by the clock of the same processor at different times, as well as the magnitudes consisting of communication delays between the sending and receiving of messages, as measured, for instance, in absolute physical time, all together considered as a set of magnitudes of the same kind, satisfy the Archimedean axiom. In physical reality it is always possible to replace a magnitude of elapsed time, of any clock or communication delay, by a corresponding magnitude of elapsed absolute physical time, thus obtaining magnitudes of the same kind. We assume a global absolute time to calibrate the individual clocks; using relative time by having the clocks send messages to one another yields the same effect - for the purposes at hand. If we do not restrict ourselves, so to speak, to Archimedean distributed systems, then the processors in the system may not have any

---

* In *Sphere and Cylinder* and *Quadrature of the Parabola* Archimedes formulates the postulate as follows. "The larger of two lines, areas or solids exceeds the smaller in such a way that the difference, added to itself, can exceed any given individual of the type to which the two mutually compared magnitudes belong". The axiom appears earlier as Definition 4 in Book 5 of Euclid's *Elements.*

sense of time. Or, they have clocks which keep purely subjective time, so that the unit time span of each processor is unrelated to that of another. That is, the set of time units is non-Archimedean if the length of every time unit is not less than a finite multiple of that of any other in the absolute global time scale. Or, the communication delays have no finite ratio among themselves or with respect to subjective processor clocks. As a consequence:

-Any process, pausing indefinitely long with respect to the time-scale of the others, between events like the receiving and passing of a message, and also any unbounded communication delay, effectively aborts activities such as an election in progress. A process can never be sure that it is the only one which considers itself elected.

-Without physical time and clocks there is no way to distinguish a failed process from one just pausing between events.

-A user or a process can tell that a system has crashed only because he has been waiting too long for a response.

Distributed systems in the sense of *physically* distributed computer networks communicate by sending signed messages and setting timers, or equivalent devices. If an acknowledgement of safe receipt by the proper addressee is not received by the sender before the timer goes off, the sender sends out a new copy of the message and sets a corresponding timer. This process is repeated until either a proper acknowledgement is received or the sender concludes that the message cannot be communicated due to failures. Thus, clocks and timeouts are necessary attributes of real distributed systems and non-Archimedean time in the system is intolerable outright. Whereas unlimited asynchrony would prevent a system from functioning properly, pure synchrony in a system cannot exist: the clocks of distinct processors drift apart in both indicated time and running speed and have to be resynchronized by algorithms running in Archimedean time as defined above.

We may call this concept of algorithms *using* physical time, instead of being *oblivious* to physical time, one of *time-driven* algorithms. The use of such algorithms would be in the area of distributed control in synchronous or asynchronous systems. Some problems necessarily have time-driven algorithms, while the algorithms for other problems may or may not be time-driven. For example, in algorithms for clock synchronization and distributed spanning tree and distributed elections, the former are time-driven by cause of their very subject matter, while the latter may be time-driven by design or not be time-driven at all. The primary goal of an investigation into the feasibility of such algorithms in [11, 12] was to demonstrate the existence of competitive time-driven algorithms with the desirable properties as mentioned. These algorithms where superior in terms of message passes. More significantly, they performed better than allowed by known lower bounds on the number of message passes required in asynchronous networks. Unfortunately, they where quite unrealistic in terms of running time. Nonetheless, we expect that genuinely more efficient algorithms than the unlimited asynchronous ones exist, in between the pure synchronous and unlimited asynchronous ones.

4. Physics

Apart from space and time, nature intrudes obstrusively in nonsequential computation in the form of physics. We give an example from the field of VLSI taken from [13].

In current chips, synchronization requirements slow down the computation to a clocked switching time, which is in the order of the delay in the longest wire. As the minimal feature width continues to decrease into the submicron range, this delay governs overall

performance more and more. In order to obtain *very high speed* integration, one way to go is to obtain a propagation delay logarithmic in the length of the wire, as in [5]. Electronic considerations show [6] that all wires then need to have the same ratio between width and length, that is, the same *aspect ratio*. Below we derive this fact, and show some of the consequences.

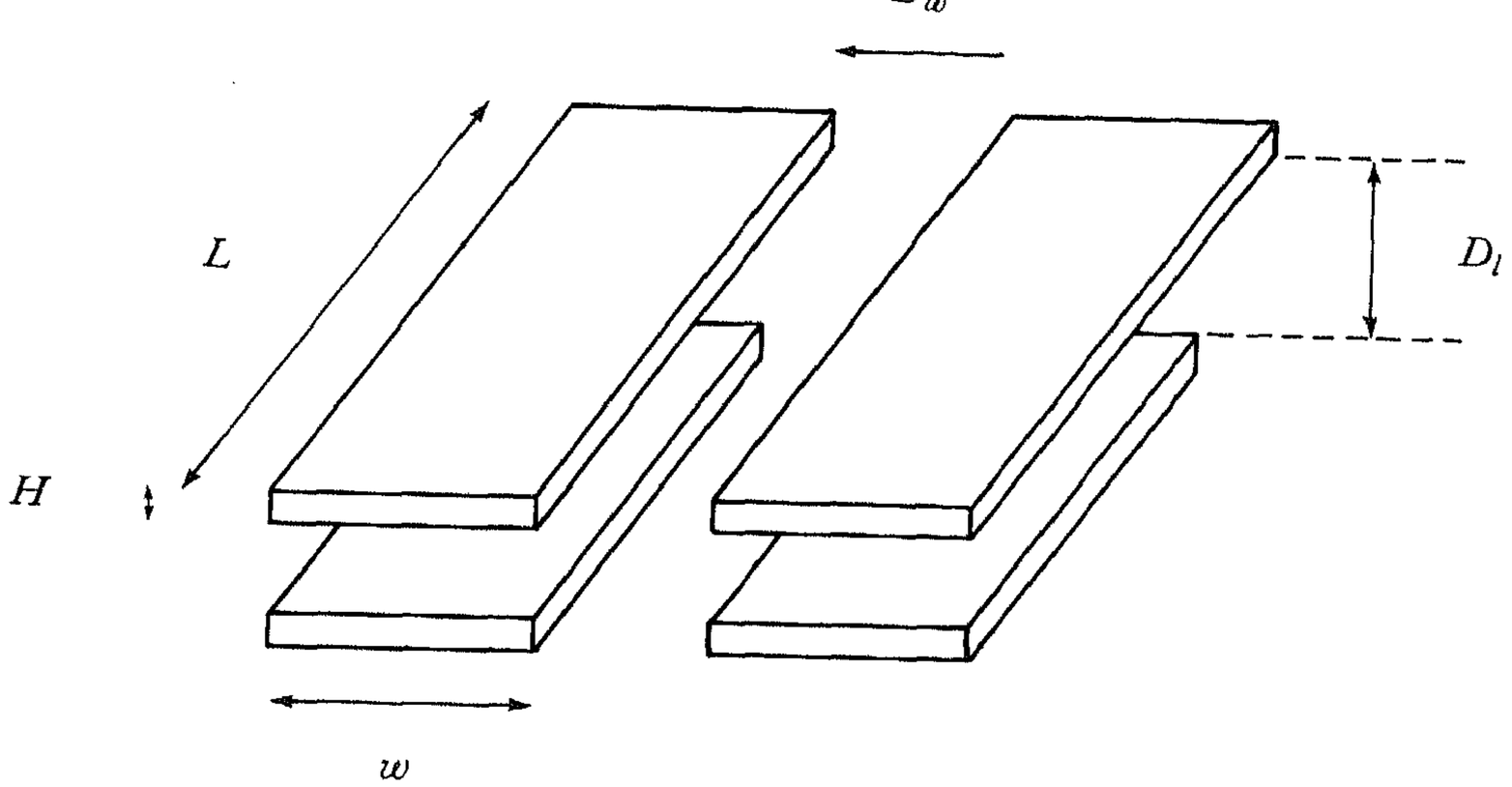
4.1. Electronics

Analysis of signal propagation delay in wires on chip requires different models in different cases: transmission line, distributed RC and lumped RC. However, the dominating factor on a densely packed chip is that a wire is not alone, but surrounded by other wires. This fact leads to the following analysis [6, 13].

The time it takes a minimum transistor to drive a wire of length L, width W and thickness H can be estimated as follows. The wire is assumed to have distance D_l to neighbouring layers and D_w to other wires in the same layer. If W_0 is the minimal width of a wire in the current technology, then the minimal transistor, consisting of a wire crossing, occupies area W_0^2. The total time T to drive a wire is approximated by:

$$T \approx (R_t + R_w)\, C_w \tag{1}$$

where R_t is the resistance of the minimum transistor, R_w the resistance of the wire and C_w its capacitance.



Therefore, the total time T can be thought of as the sum of the time T_d needed to drive a zero resistance wire of capacitance C_w, and the time $R_w C_w$ needed to transport the appropriate charge from a zero resistance source. Roughly, T_d is the time needed to transport the necessary charge through the bottleneck consisting of the switch (the minimal transistor), and $R_w C_w$ is the time needed to distribute the charge appropriately over the wire w. Since the resistance of a wire is proportional to its length and inversely proportional to its cross section we have:

$$R_w = \rho_w \frac{L}{WH} \tag{2}$$

where ρ_w is the resistivity of the considered wire material. The capacitance of a wire is inversely proportional to the distance of its neighbouring wires and layers, and proportional

to the area of the side facing that neighbouring layer or wire:

$$C_w = 2\epsilon_w L \left(\frac{H}{D_w} + \frac{W}{D_l} \right) \tag{3}$$

where ϵ_w is a proportional constant consisting of the product of the permitivity of free space and the dielectric constant of the insulating material (usually SiO_2). Thus,

$$R_w C_w = 2\rho_w \epsilon_w \frac{L^2}{WH} \left(\frac{H}{D_w} + \frac{W}{D_l} \right) . \tag{4}$$

This suggests a signal propagation time quadratic in L. However, the resistance R_t of the minimum transistor dominates in (1) for the magnitudes of L under consideration (smaller than, say, 1 foot). We can decrease that term by fitting a larger driver transistor to the wire. This transistor, in its turn, must be driven by the minimal transistor. Iterating this scheme, cf [5], we obtain a sequence of transistors, of which each next one is a factor α larger than the preceding one. The final transistor in the sequence should be large enough to drive the wire in a sufficiently short time. (We can think of this scheme as a sequence of switches where each switch serves to switch the next larger switch, and the largest switch in the sequence controls the large channel through which the charge is transported to the wire. Although the time to actually pass the appropriate charge from source to wire can be made smaller by fitting a larger final driver transistor to the sequence, there seems no way to get rid of the time needed to switch all transistors in between the smallest transistor and the largest one.) The time to drive a driver with capacitance C_2 by a driver with smaller capacitance C_1 is given by [5]:

$$\tau \frac{C_2}{C_1} \tag{5}$$

where τ is the time it takes a minimal transistor to charge the gate of another minimal transistor. If C_t is the capacitance of the minimal transistor then for a ramp of r drivers:

$$r = \log_\alpha \frac{C_w}{C_t} \tag{6}$$

taking $T_d = r\,\tau\,\alpha$ time to charge the wire if it had no resistance. The capacitance of the minimum transistor is given by

$$C_t = \epsilon_t \frac{W_0^2}{D_0} , \tag{7}$$

where D_0 is the thickness of the gate insulator and ϵ_t is the product of the permitivity of free space and the dielectric constant of the gate insulator. Thus we can drive a zero resistance wire of capacitance C_w through a sequence of r drivers for fixed α in time:

$$T_d = \alpha\,\tau \log_\alpha \frac{C_w}{C_t} . \tag{8}$$

>From (1), (4) and (8) we obtain an expression for $T = T_d + C_w R_w$. In [6] it was observed that by keeping the derivatives, with respect to L, of the two terms T_d and $C_w R_w$ balanced:

$$\frac{\alpha\,\tau}{L \ln\alpha} \approx \rho_w \epsilon_w \frac{L}{WH} \left(\frac{H}{D_w} + \frac{W}{D_l} \right) \tag{9}$$

T grows logarithmic in L. Viz., by assumption of equality (10) we obtain:

$$T \approx \frac{\alpha\tau}{\ln \alpha} \left\{ \ln \left[\frac{\epsilon_w D_0 L}{\epsilon_l W_0^2} \left(\frac{H}{D_w} + \frac{W}{D_l} \right) \right] + 1 \right\}$$

According to (9) we obtain logarithmic signal propagation delay by, all other things being equal,

$$L^2 \left(\frac{1}{W D_w} + \frac{1}{H D_l} \right) = \text{constant} \qquad (10)$$

rather than by just keeping L^2 proportional to WH as in [6]. Keeping the interwire distance proportional to the wire width, and the interlayer distance proportional to the wire height, we observe that if W, H and L are kept in proportion a logarithmic propagation delay is attained. (Note that we cannot reach this effect by keeping the wire width the same but using very 'tall' wires or vice versa.) The *aspect ratio* of a wire is the quotient of its width and length. To obtain a logarithmic signal propagation delay we thus need the fixed constant aspect ratio following from (9) and (10) for all wires in the layout. In designing such a high speed layout we therefore need to install drivers to drive the long wires and to design all wires with a constant aspect ratio $a > 0$. Therefore, a wire of length L in such a layout has *area* aL^2. The area taken by the driver is linear in the length of the wire [6]: the minimal transistor occupies area W_0^2, the next driver area αW_0^2, and so on for $\log_\alpha L$ terms for an L-length wire. The total driver area for an L-length wire becomes $W_0^2 (L-1)/(\alpha-1)$. This area is required at the lowest silicon layer of the chip; the long interconnect wires are executed in the upper metal layers.

The effect of having all wires in the layout with the same constant aspect ratio spells disaster for circuits which necessarily have many long wires. This holds for trees, but more so for fast permutation networks. However, let us look what happens for natural wire length distributions.

4.2. Wire Length Distributions

Let $f: \mathbb{N} \to \mathbb{N}$, connected with a VLSI layout, be a *wire length distribution* function which yields the number $f(i)$ of wires of length i in the design.

Every VLSI layout must have a constant bounded fan-in and fan-out of wires for the components (transistors). If the chip area is A, then a reasonable assumption is that the maximal wire length on a chip does not exceed

$$L_{\max} = \sqrt{A} \quad . \qquad (11)$$

Consequently, the amount of wires in the layout is given by

$$\#\text{wires} = \sum_{i=1}^{\sqrt{A}} f(i) \quad . \qquad (12)$$

To achieve logarithmic propagation delay we can estimate and bound the layout area occupied by the fattened wires as follows. Let C be the amount of area of the layout occupied by *non-wire components* such as transistors. Assuming that C is also the order of magnitude of the number of basic components like transistors or logic gates in the circuit we can reason as follows. Since the wires only serve to connect components we have $C \in O(\#\text{wires})$ in a connected layout. The components are assumed to have at most a

limited t connections to attach wires, which we suppose to account also for the *fan-in* and *fan-out* of the interconnect wires. Therefore $C \in \Omega(\#\text{wires})$ and consequently $C \in \Theta(\#\text{wires})$. Since we are primarily interested in orders of magnitude in the sequel, we are justified to use C interchangeably for the amount of area occupied by the non-wire components, the number of non-wire components and the number of wires. The maximal area occupied by the wires (and interwire distances) under (10) is bounded by the available area:

$$\sum_{i=1}^{\sqrt{A}} f(i)\, a i^2 \approx A - C \quad , \tag{13}$$

where a is the constant quotient of width and length (the *aspect ratio*) of the connect wires as required by (10). Using a simple theoretical argument and an experimental study of actual layouts [2] develops the following wire length distribution relationship:

$$f(i) = \lfloor c i^{-\lambda} \rfloor \ (1 \leqslant i \leqslant L_{\max}) \text{ and} \tag{14}$$

$$f(i) \approx 0 \ (i > L_{\max})$$

for a *normalization* constant c yet to be chosen. Here $L_{\max}$ is a constant related to the size of the array (rectangular chip) and the adequacy of the placement; and λ is a constant characteristic of the logic. Equation (14) is derived using "Rent's Rule" which states that the average number of terminals per complex of C elements (in units, modules, cards, gates etc.) is tC^p, where t is the number of connections per individual element and p is the Rent constant characteristic of the logic complex. The analysis goes by dividing a square array of cells into 4 equal square arrays recursively down until the individual areas are the individual elements of the original logic. On each level of the recursion the number of connections crossing boundary lines is determined using Rent's rule. This shows that $\lambda \approx 3 - 2p$. In [2] experimental results are given for some actual layouts placed using a hierarchical placement program: layouts for high-speed logic where p was found to be 0.75 and a layout for a hand calculator chip with $p = 0.59$. Let furthermore the network be connected, so the maximal amount of area units C available to place the components is not greater than the number of wires plus 1.

Considering just the wire length distribution while leaving free the actual circuit topology, placement and routing in the layouts, attaining a logarithmic signal propagation delay by changing constant wire width to constant aspect ratio for all wires in a layout can carry a surprisingly severe penalty. This follows immediately from (11), (12), (13) and (14), and is expressed by the theorem below. The (simple) analysis of this fact, and the proof of the Theorem, are relegated to the Appendix.

Theorem. *Let the original layout area be A and the original amount of wires in the layout be C. For the wire length distribution $f(i) = \lfloor c i^{-1} \rfloor$ for $1 \leqslant i \leqslant \sqrt{A}$ and $f(i) \approx 0$ for $i > \sqrt{A}$, the change from constant wire width to wires with a constant aspect ratio has the following effect.*

(i) *Keeping f and C the same, the area has to increase from A to* $\exp(\Omega(\sqrt{A}))$.

(ii) *Keeping f and A the same, the number of wires (c.q. components) has to decrease from C to* $O(\log C)$.

(iii)*Keeping A and C the same, the wire length distribution has to change to* $f'(i) = \lfloor c'i^{-(2+\epsilon)} \rfloor$ *for some small* $\epsilon > 0$ $(1 \leqslant i \leqslant \sqrt{A})$.

We observe that in case (i) of the Theorem the wires get so long that the logarithmic propagation delay turns out to yield about the same absolute time delay as in the original wires. In case (ii) of the Theorem matters are probably as bad because the bit capacity of the chip has been logarithmically reduced. Finally, in case (iii) of the Theorem the subject circuit topology may not have a layout with the required wire length distribution.

It therefore appears that only circuits for which there are layouts with wire length distributions with relative large λ, will profit from this scheme for logarithmic signal propagation delay.

Acknowledgement.
Baruch Awerbuch, Evangelos Kranakis and Yoram Moses read the draft and gave advice on presentation.

References

[1] Burks, A.W., H.H. Goldstine, and J. von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument", Report, Princeton Institute for Advanced Study, June 1946.

[2] Donat, W.E., "Wire length distribution for placement of computer logic," *IBM J. Res. Develop.*, vol. 25, pp.152 - 155, 1981.

[3] Goldstine, H.H., *The Computer: from Pascal to von Neumann.* Princeton, N.J.:Princeton University Press, 1972.

[4] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the Assoc. Comp. Mach.*, vol. 21, pp.558-565, 1978.

[5] Mead, C. and L. Conway, *Introduction to VLSI Systems.* Reading, Mass.:Addisson-Wesley, 1980.

[6] Mead, C. and M. Rem, "Minimum propagation delays in VLSI," *IEEE J. on Solid State Circuits*, vol. SC-17, pp.773 - 775, 1982. Correction: *Ibid*, SC-19 (1984) 162.

[7] Mullender, S.J. and P.M.B. Vitányi, "Distributed match-making for processes in computer networks," pp. 261-271 in Proceedings 4th Annual ACM Symposium on Principles of Distributed Computing (1985).

[8] Neumann, J. von, "First draft of a report on the EDVAC", Draft Report, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, May, 1945.

[9] Turing, A.M., "On computable numbers with an application to the Entscheidungsproblem," *Proc. London Math. Soc.*, vol. 42, pp.230-265, 1936. Correction, *Ibid, 43* pp. 544-546 (1937).

[10] Ullman, J.D., *Computational Aspects of VLSI.* Rockville, Maryland:Computer Science Press, 1984.

[11] Vitányi, P.M.B., "Distributed elections in an Archimedean ring of processors," pp. 542-547 in Proceedings 16th Annual ACM Symposium on Theory of Computing (1984).

[12] Viťanyi, P.M.B., "Time-driven algorithms for distributed control", Report CS-R8510, Centre for Mathematics and Computer Science, Amsterdam, April, 1985.

[13] Vitányi, P.M.B., "Area penalty for sublinear signal propagation delay on chip," in Proceedings 26th Annual IEEE Symposium on Foundations of Computer Science (1985).

Appendix

>From (13) and (14) we can estimate the maximal figure for the normalization constant c. For $\lambda \neq 3$:

$$c \approx \frac{(A-C)(3-\lambda)}{a(A^{(3-\lambda)/2}-1)} \quad , \tag{15a}$$

and for $\lambda = 3$,

$$c \approx \frac{2(A-C)}{a \log A} \quad . \tag{15b}$$

Consequently, for $\lambda \neq 1$ & $\lambda \neq 3$ by (12):

$$C \approx \sum_{i=1}^{\sqrt{A}} f(i) \approx \frac{(A-C)(3-\lambda)(A^{(1-\lambda)/2}-1)}{a(1-\lambda)(A^{(3-\lambda)/2}-1)} \quad . \tag{16a}$$

and for $\lambda = 3$,

$$C \approx \frac{(A-C)(A-1)}{aA \log A} \quad . \tag{16b}$$

For $\lambda = 1$,

$$C \approx \sum_{i=1}^{\sqrt{A}} f(i) \approx \frac{A-C}{a(A-1)} \log A \quad . \tag{16c}$$

(Note: for $\lambda < 1$ we obtain $c < 1$, resulting in $f(i) \approx 0$ also for small i, and C a small constant.)

For comparison we give an analogous analysis under the constant wire width assumption. Then equations (11) - (12) stay the same but equation (13) becomes

$$\sum_{i=1}^{\sqrt{A}} f(i)\, i \approx A - C \quad . \tag{17}$$

Thus, for $f(i) = \lfloor ci^{-\lambda} \rfloor$ $(1 \leqslant i \leqslant \sqrt{A})$ and $f(i) \approx 0$ $(i > \sqrt{A})$ and with A, C and c as above we obtain the following relations. For $\lambda = 1$:

$$c \approx \frac{A-C}{\sqrt{A}-1} \tag{18}$$

$$C \approx \frac{(A-C)\log A}{2(\sqrt{A}-1)} \quad .$$

For $\lambda \neq 1$ & $\lambda \neq 2$:

$$c \approx \frac{(2-\lambda)(A-C)}{A^{(2-\lambda)/2}-1} \tag{19}$$

$$C \approx \frac{(2-\lambda)(A-C)(A^{(1-\lambda)/2}-1)}{(1-\lambda)(A^{(2-\lambda)/2}-1)} \quad .$$

For $\lambda = 2$:

$$c \approx \frac{2(A-C)}{\log A} \tag{20}$$

$$C \approx \frac{2(A-C)(\sqrt{A}-1)}{\sqrt{A} \log A} \quad .$$

(Note: for $\lambda < 0$ we obtain $c < 1$.) For $\lambda > 0$ we have $C \in \Omega(\sqrt{A})$. Thus:

Proof of Theorem. Since we assume the circuit to be connected we have $A > A - C > A/2$ in the various equations. We also assume $A \gg 1$.

(i) Equate expression (18) for C with expression (16c) for C, with A' substituted for A in the latter. This yields $\log A' \in \Omega(\sqrt{A})$.

(ii)Substitute C' for C in equation (18) and express C' in terms of C by eliminating A from the resulting equation and (16c).

(iii)Equate expression (18) for C with expression (16a) for C (expressions (16b) and (16c) contradict (18)). The terms $(A - C)$ on both sides cancel each other. Solving λ yields $\lambda = 2 + \epsilon(A, a) > 2$ with $\epsilon(A, a) \rightarrow 0$ for $A \rightarrow \infty$ and a constant. Every distribution with exponent equal or larger than this λ suffices. ●