NONCONVENTIONAL COMPUTERS

Today, a "computer," without further qualifications, denotes a rather well-specified kind of object; we will consider a computer "nonconventional" if its physical substrate or its organization significantly departs from this de facto norm. Thus, the thousands of literate Greeks that ended up in Rome as secretaries and accountants after the "liberation" of Greece in the second century B.C. would be viewed today as nonconventional computers, even though at that time one certainly couldn't imagine a more ordinary kind of personal computer.

Furthermore, we will be more concerned with features that ultimately have to be answerable to physics (the mechanisms by which the logic elements operate, the geometry of interconnection, the overall flow of energy and information) than with architectural variants of a "firmware" nature (reduced instruction set, speculative execution of program branches, etc.).

Think of an indefinitely extended prairie. If you drop a match, fire will spread outwards in a roughly circular front. Owing to random irregularities in propagation speed (because of varying grass thickness, flammability, etc.) the shape of the burning front will eventually become fairly irregular. Since the grass is quickly consumed, fire cannot linger or come back the way it came: It must move on. However, under the steady pumping of solar energy, in a few weeks grass will regrow and fire will be able to return to an already visited region; Fig. 1 shows characteristic propagation patterns. A substrate that supports spontaneous activity of this kind is called an *excitable medium.*

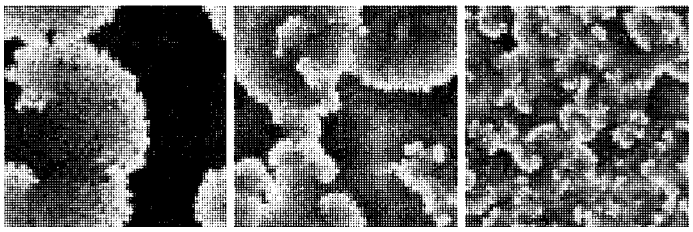
Figure 1. Propagating fire front patterns, at different scales (from a computer simulation).

The activity we have seen is complex but chaotic. Can this complexity be disciplined without postulating even more complex agents? Let us consider passive fire walls. One can create fire corridors with two parallel walls a few hundred feet apart and extending for hundreds of miles. From a distance, the fire front propagating along a corridor will look like a localized pulse traveling along a wire at a well-characterized speed. If this wire makes a closed loop, a pulse will recirculate indefinitely. On a T-junction between wires, a signal coming along one branch will fan out along the other two branches. Thus, a loop with a tap (a σ-shaped circuit) will, once primed, act as a clock, sending out pulses at regular intervals.

It is not hard to make a wire constriction that will let fire go through in only one direction—a "rectifier." Moreover, a signal will not propagate through a section of wire that has recently been visited by another signal; thus, a signal sent with appropriate timing can inhibit or "lock out" another signal. Indeed, using just prairie fire and passive walls, one can construct on a majestic scale a fairly close approximation of a network of neurons and axons, or even a digital computer.

In principle, all that is needed to make a computer is an excitable medium and a way to channel the propagation of activity in it-the rest is detail. Here we shall examine significantly and often strikingly different ways to fill in this "detail." Besides providing an instructive record of past evolutionary struggles, nonconventional schemes of computation contribute to that *rich reservoir of genetic variability* that has put computers at the forefront of evolution.

The proceedings volume (1a) provides a representative sample of recent ideas in this area.

BASIC SETTING

Computation Universality

The essence of computation is that a mechanism displaying arbitrarily complex behavior can be constructed without making recourse to ever more complex components: We just need to increase the *number* of parts, not the *complexity* of the individual parts. Minsky (1) provides a solid and widely accessible introduction to these concepts.

Consider a catalog of building blocks, or *elements,* each capable of computing some simple function, and such that any output of one element can be used as an input by any other. (For instance, in the heyday of analog computers the usual convention was that outputs should produce voltages in the ± 10 V range and inputs should accept any voltage in that range.) Provided that the catalog assortment satisfies certain minimum prerequisites, any function that can be computed by a mechanism *no matter how complex* can also be achieved merely by composing elements chosen from that catalog. (In the case of analog mechanisms, “achieved” is understood to mean “to any desired degree of approximation”). For example, if the catalog lists just the logic functions (or logic “gates”) AND, OR, and NOT (Fig. 2), then it can be proved that *any* logic function with a finite number of inputs can be put together from items picked from that catalog. In this sense, we say that these three elements constitute a *universal set of logic primitives.* (At the cost of slightly more cumbersome constructions, one can make do with an even more restricted catalog, containing as a single element the NAND gate, also shown in Fig. 2.)

A related but more general concept, which arises when we are dealing with indefinitely extended computing tasks, is that of *computation universality.* Not only can we do arbitrarily complex computation using only simple logic elements, but we do not even need an arbitrarily large *assembly* of them, because that can be simulated by a *Turing machine* consisting of

- A *finite* assembly of active elements, given once and for all (the “head”).
- An indefinitely-extended, *passive* storage medium (the “tape”).
- A finite *description* of the (possibly infinite) machine we have in mind (the “program”). This may reside on the tape.

Intuitively, the head can be “time-shared” so as to perform under the guidance of the program all the functions of the target machine, using the tape to keep track of “who was doing what to whom.” It turns out that extremely simple head-and-tape structures [i.e., with few states for the head and few symbols for the tape alphabet (1)] are already capable of doing, in this fashion, anything that can be done by more complex structures. Given that computation universality is so easy to attain, when we say “computer” without further qualifications we shall mean machinery that does have this property.

AND		OR		NOT		NAND	
in	out	in	out	in	out	in	out
00	0	00	0	0	1	00	1
01	0	01	1	1	0	01	1
10	0	10	1			10	1
11	1	11	1			11	0

Figure 2. The AND, OR, NOT, and NAND logic elements (or logic “gates”); the symbols 0 and 1 represent the logic values “false” and “true.” The first three elements make up a *universal set of logic primitives;* the fourth element by itself constitutes such a set.

Digital Versus Analog Devices

When active devices (e.g., tubes or transistors) were an expensive resource, it appeared wasteful to devote a few of them just to making a simple logic element such as a gate or a flip-flop when they could be used for more sophisticated mathematical functions. For instance, by bringing together to a summing node a number of resistors one can compute the weighted sum of several real variables (Fig. 3); what’s more impressive, a matched transistor pair can be used to accurately compute logarithms or exponents with a 10^8 dynamic range! (In either case, the processing stage must be followed by an isolation amplifier.) In fact, analog circuitry seriously vied with digital circuitry in the early days of scientific computing. Today, analog circuitry is still competitive in certain specialized real-time tasks such as TV signal processing. Even there, though, it is gradually being taken over by *digital signal processors*—small computers that specialize in fast numerical computation.

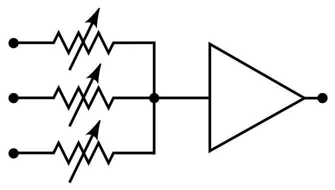
Figure 3. Analog adder. By using resistors of different values, the terms of the sum can be given different weights. The summing stage is completed by a voltage buffer for isolation.

A digital element handles binary variables. If, using comparable physical resources, an analog element can handle *real-valued* variables, doesn’t it have, in a sense, “infinitely more” computing power? There is no doubt that an analog simulation of a continuous system may in certain cases outperform a digital simulation of it (e.g., one done by a floating-point processor). This point has been well-argued by Mead (2). What should be clear, however, is that the two approaches are *equivalent* in terms of computing power; that is, either approach can simulate the other to within a *constant factor* in terms of storage capacity and processing speed. In fact, because of thermal noise and fabrication tolerances, the nominally continuous range of an analog variable is actually equivalent to a modest number of *distinguishable* states. Moreover, when one changes one of the inputs in Fig. 3, the new voltage at the summing node is approached exponentially with a time constant τ_{analog}; to achieve a precision of k significant digits, one must wait for a time $\approx k\tau_{\text{analog}}$. If the same input data were encoded as binary strings and processed by a serial digital adder with a clock period τ_{digital}, one would get k digits in a time $\approx k\tau_{\text{digital}}$. Thus, contrary to claims that are occasionally made, analog computers do not hold the key to capabilities that transcend those of digital computers. (But see the section entitled “Quantum Computation” for a remarkably novel approach to this issue.)

For the rest of this article we shall restrict our attention to digital computation unless explicitly noted.

Serial Versus Parallel Processing

A computer must be able to deal with indefinitely large amounts of information. Conventional computers process this information *serially,* in the sense that there is a single, localized, active piece of machinery through which data must sequentially stream in order to interact and be transformed: In the Turing machine, the head moves back and forth along the tape, reading data from it and writing new data back to it. In ordinary computers, the active unit, or central processing unit (CPU), is stationary, and it is the data that do the moving. In practice, a sizable amount of data is kept in a random access memory (RAM) array that is optimized for fast random access; to access a RAM location the CPU specifies its address, and some ancillary active circuitry transports the corresponding data; this is the essence of the *von Neumann architecture* (the *Harvard* architecture is similar, but keeps separate memory banks for program and data). Larger amounts of data are kept on storage media, such as magnetic disk or tape, that are served by more rudimentary transport resources and typically allow only sequential access to the data.

In many circumstances it is desirable to have a number of related data-processing operation take place concurrently—an approach that is loosely termed *parallel* computation.

1. As one comes close to the limits of a technology, the cost of faster machinery grows out of proportion to the attendant speed gain. For demanding computational tasks, it may be more cost-effective to use a "fleet" of slower processors rather than a single ultra-high-speed unit.
2. Certain computational tasks, such as the simulation of spatially extended physical systems (weather forecasting, materials science, brain modeling), are intrinsically parallel. The evolution of a site is immediately affected only by its neighbors, that is, the sites directly connected to it; therefore, in the short term, distant sites can be updated at the same time without reference to one another, and thus by separate processors.
3. Time-sharing a single processor between a number of sites may entail substantial overhead. Data from a site's neighborhood are typically copied into the processor's internal registers for efficiency in processing. When the processor's focus is moved from one site to another, these data have to be saved and new data loaded. Using a dedicated processor for each site eliminates this overhead.
4. In a finite-difference scheme (as may arise from discretizing a differential equation) a site typically contains several floating-point variables, and its updating entails a number of algebraic, transcendental, and address-manipulation operations. In finer-grained models such as lattice gases, the number of sites may be several orders of magnitude larger, while the updating of a site may involve just a few logic operations on a few bits. On this kind of task, most of the resources of a conventional processor would be wasted. For the same amount of resources, a better approach is to use an array of thousands of microscopic site processors.
5. Finally, a world consisting of a finite active head and an indefinitely extended passive tape is only an approximation. In real life, most of the data that a processor will see during a computation are not actually present in a storage medium at the beginning of the computation, but will be deposited there by other agents as the computation progresses (think of an airline reservation system). A collection of loosely interconnected processors provides a better paradigm for this arrangement.

Fine Grain versus Coarse Grain

Much parallelism in computation is achieved today by loosely networking or more tightly coupling a modest number of conventional processors (3); in the latter case, the connectivity is often provided by having all processors share a single memory. In either case, each node "feels," at least in the short term, much like a conventional von Neumann machine—with a sizable processor running a large instruction set and having access to a large expanse of data.

In an attempt to achieve a better match either with the nature of a problem or with the physics underlying the hardware, many nonconventional schemes of computation adopt a much more finely subdivided architecture, where the number of processors is large but each has a limited scope. In such *fine-grained* architectures, task coordination between the processors may explictly be achieved by some centralized form of control or, more implicitly, by prearranging the individual nodes' nature and their interconnection pattern so that this pattern itself constitutes the program (4). One may even employ identical nodes and uniform (or uniformly random) interconnection, with no external control, and effectively encode the program in the pattern of initial data; this approach, used in programmable-logic arrays and field-programmable gate arrays, is commercially viable and is gaining popularity.

An intermediate approach is *configurable computing* (5), where the interconnection between small but self-contained functional blocks can be reconfigured in real time, so as to have "just in time" hardware.

Microscopic Law Versus Emergent Behavior

An even more extreme form of "laissez faire" is when not only the network is fine-grained and uniform, but the initial data are random (at least on a short scale). In this case, the behavior that emerges can only be the macroscopic expression of the microscopic law built into the node—that is, is an *attractor of the the dynamics* (6). Though the attractors are in principle completely determined by the microscopic dynamics, their specific form is not easibly deducible from it; the whole point of the computation is to make the attractors manifest (see section entitled "Associative Networks" and Fig. 11).

In terms of applications, emergent computation is relevant to statistical mechanics, materials science, economics, voting theory, epidemiology, biochemistry, and the behavior of social, swarming, and schooling species (7–9).

Polynomial Versus Exponential Connectivity

Many common problems are of *exponential* complexity, in the sense that the computational resources needed to solve the problem grow exponentially with the size of the input data. This can be easily seen as follows. To determine the fitness of an "organism" in a given environment, the general method is to run a simulation of the entire system and in this way directly evaluate the desired fitness function (number of offspring, market share, etc.). If the organism consists of n parts, the cost of one simulation run will typically be polynomial in n, because both the size of the simulation (number of variables) and its length (number of time steps) will grow in proportion to n. Suppose now that several variants are available for each of the parts (in a gene, for instance, there are four choices for each base pair). To find the best combination of parts, the general procedure is to determine, by simulation, the fitness of all the possible combinations; the number of these is exponential in n. Thus, while simulation is typically a polynomial task, optimization is typically *exponential.*

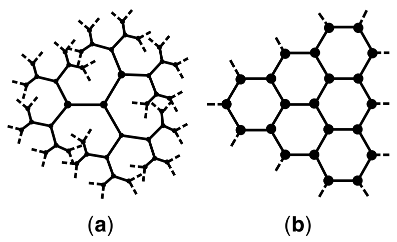

Figure 4. (a) In this tree network, the number of nodes reachable in n steps grows as 2^n (exponential growth). (b) In this mesh network, it grows as n^2 (polynomial growth).

A naive way to satisfy exponential computing demands is to design a parallel computer with exponential interconnection—that is, a computing network in which the number of sites that can be reached from a given site in n steps grows exponentially with n [for instance, in the tree of Fig. 4(a) the number of new sites one can reach starting from any site doubles with each step]. Such a network, however, cannot be conformally embedded in three-dimensional physical space. Even if one could actually provide an exponential number of processors, the interconnection geometry must be drastically deformed to fit into three-dimensional space; nodes that logically are separated by a single link will have to be spaced further and further apart, and communications will slow down accordingly.

Of course, in order to simplify high-level programming it is often convenient to *simulate* an exponential architecture on a conventional computer; this is essentially the route offered by the LISP programming language (10). In the 1980s, an architecture optimized for this kind of deception—the LISP machine—enjoyed brief popularity in the Artificial Intelligence milieu.

A related multiprocessor arrangement is the *hypercube*—a network whose growth is exponential in the short term but then tapers down and actually converges to a finite size. In a d-dimensional hypercube a node has d first neighbors, about $d/2$ second neighbors, and, in general,

$$\binom{d}{n}$$

nth neighbors. Since the hypercube's vertices are in a natural one-to-one correspondence with all the possible states of a binary string of length d, a hypercube is a good architecture for problems of combinatorial optimization. Even though ordinary space has three dimensions, a 16-dimensional hypercube, with 64K sites, can conceivably be "folded" onto a printed-circuit board; in fact, hypercube "accelerator cards" enjoyed a brief success. However, one must bear in mind that, while going from a 16-bit microprocessor to a 32-bit one is a comparatively modest increment, going from a 16-dimensional hypercube to a 32-dimensional one is a tall order, because the latter would have four billion sites and 64 billion links! In this sense, the hypercube architecture is not *scalable.*

Various interconnection topologies are discussed in (11). The extent to which the physical geometry of a network can be ignored in favor of its logical organization depends, of course, on the ratio between processing time (activity at a node) and communication time (travel between logically adjacent nodes). When this ratio is high, the actual geometry has little relevance; this is the case of *intranet* architectures (collections of workstations connected by a local-area network) and, to a great extent, of the Internet itself. In fact, we are witnessing the birth of a commodity market for large packets of CPU cycles. For applications that are computation rather than communication-intensive, it matters little *where* these packets are executed; thousands of disparate computers scattered all over the world may be successfully harnessed to work on a single task (12).

Conversely, intersite communication looms large in fine-grained computational tasks, where data go through a node almost instantaneously. Here, the most efficient architectures tend to directly reflect the polynomial interconnectivity of physical spacetime, and ideally one has a polynomial-growth network, or *mesh,* directly embedded in physical space, as in Fig. 4(b). (As stressed in the section entitled "Cellular Automata Machines" there are other practical factors besides interconnection geometry that one must take into account in the design of a viable fine-grained multiprocessor.)

MIMD Versus SIMD Architectures

A basic dichotomy in conventional parallel computers is between MIMD architectures (multiple instruction stream, multiple data stream) and SIMD (single instruction, multiple data), according to a classification proposed by Flynn (13). An extreme case of MIMD is a network of ordinary computers running different programs related to the same task, with just enough synchronization to ensure that subtasks are carried out in the appropriate order. A typical example of SIMD is a *vector processor,* where all elements of a "vector" (an array of numbers, a pixelized image) are subjected in parallel to one processing step after another.

The distinction between SIMD and MIMD does not properly apply to structures like neural networks or cellular automata (see below), where the atomic processors are not governed by an instruction stream, but each continually applies a *fixed,* built-in transition function or transfer function to the incoming data. (In a cellular automaton this function is the same for all cells, while in a neural network each node may have been programmed with a different set of input weights and, typically, with a different interconnection pattern).

NEURAL NETWORKS

Neural networks (14) are circuits consisting of a large number of simple elements, and designed in such a way as to significantly exploit aspects of collective behavior—rather than rely on the precise behavior of the individual element.

In spite of their enormous speed, conventional digital computers compare poorly in many tasks with the nervous system of animals. How much of the architecture of a nervous system does one have to reproduce in order to capture the strong points of its behavior? Historically, neural networks were proposed as an alternative type of computing hardware, loosely patterned (both in the nature of the circuit elements and in the way they are interconnected) after the animal nervous system. Today, however, it is becoming clear that rather than representing just another type of *computing medium,* neural networks represent a different *conceptual approach* to computation, depending in an essential way on the use of statistical concepts. In this sense, the theory of neural networks plays in information processing a role analogous to that of statistical mechanics in physics. We are no longer thinking so much in terms of a distinguished kind of hardware as of a distinguished class of *algorithms;* as a matter of fact, many neural-network applications are routinely and satisfactorily run on ordinary digital computers.

A typical application for neural networks is to help in making decisions based on a large number of input data having comparable *a priori* importance—for instance, identifying a traffic sign (a few bits of information) from the millions of pixels of a noisy, blurred, and distorted camera image. In general, the neural-network approach seems best suited to computational problems of large width and moderate depth—“democratic” rather than hierarchical algorithms. Note that segmentation of connected speech into words—which is a hard task for conventional computers—is performed by our brain with a latency of just a fraction of a second, and thus cannot involve more than a few levels of neurons.

Neural-network design and analysis typically assume a regime of high hardware redundancy: It then becomes both possible and desirable to program a network for a given task by indirect methods (training by example, successive approximations, simulated annealing, etc.). Indeed, the metaphor of a network “learning” its task instead of being “programmed” for it is one of the most appealing—and elusive—aspects of this discipline. By empirical means, it is not hard to come up with a neural-network design that works for a certain toy problem; it is much harder to prove the correctness of the design and rationally determine its potential and limitations. The importance of theoretical work in this context cannot be overstated.

Abstract Neurons

The human brain consists of about 10^{11} neurons of various types; each neuron typically connects, via an axon that eventually branches out into strands and substrands, to many thousand neurons. The firing of a neuron is mostly an all-or-nothing business; this *discrete* character is retained as the pulse travels down an axon. However, upon arrival to a destination neuron the pulse is handled by a synaptic interface characterized by an *analog* parameter (typically, an excitation or inhibition weight) whose value may be to some extent history-dependent. The complete physiological picture is rather complex.

A drastically simplified model of a neuron, proposed by McCulloch and Pitts (15), is shown in Fig. 5. The neuron can be in one of two states, $+1$ and -1, which may be thought of

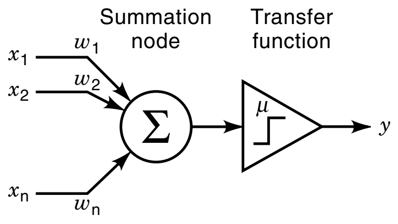

Figure 5. McCulloch–Pitts neuron. The summation node constructs the weighted sum (with coefficients w_1, w_2, . . .) of the inputs. Depending on whether or not this sum exceeds a threshold μ, an output of $+1$ or -1 is returned by the transfer function.

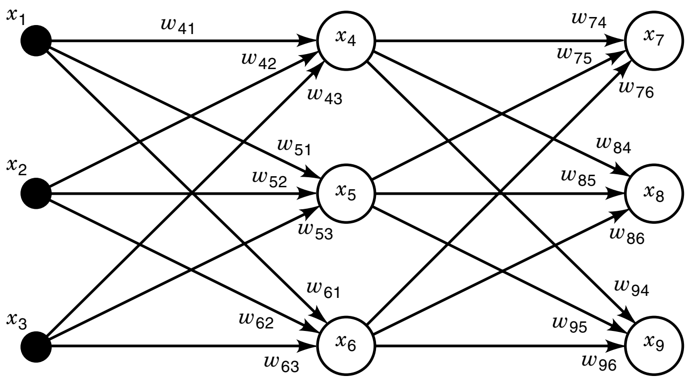

Figure 6. Two-layer perceptron. The black nodes denote external inputs.

as “on” and “off,” or “true” and “false”; this state appears at the neuron’s output. The inputs may come from other neurons or from external stimuli. State updating may be *synchronous* (all neurons are updated simultaneously at times $t = 0, 1, 2,$ . . .) or *asynchronous* (each neuron is updated at random times with a given probability per unit time). The new state of the neuron is determined by the inputs as follows. Input x_j is multiplied by a weight w_j, representing the strength of the corresponding synaptic connection (positive weights correspond to *excitatory* synapses, negative weights correspond to *inhibitory* ones). The contributions from all inputs are added and compared with a threshold μ; the neuron turns on if the threshold is exceeded.

The McCulloch–Pitts neuron is a universal logic primitive (1); for instance, with a two-input neuron, weights of -1 for each input, and a threshold of $-1/2$, the neuron will continually fire unless at least one of the inputs is turned on, thus yielding the NAND function. But why, then, not use ordinary logic elements to begin with? The answer is that the neuron is optimized for a different kind of architecture, where a single node may have thousands of inputs (as in the human brain) rather than just a few. An arbitrary logic function of that many inputs would consist of lookup table of astronomical size (i.e., exponential in the number of inputs); to have an element that responds in a nontrivial way to all of its inputs, but whose complexity grows only proportionally to the number of inputs, one must drastically restrict the nature of the interaction. In the neuron this is achieved by the two-stage design of Fig. 5, namely, a summation node followed by a transfer function. The first stage deals with *all* the inputs, but only in an *additive way;* while the second stage, which has only one argument, contributes the *nonlinear* response which is essential for computation universality.

Developments

As we’ve seen, neural networks started out as an exercise in mathematical biology. The first networks to systematically use many-input neurons were the *perceptrons* (16), in which neurons are arranged in regular layers, with no feedback from a layer to previous ones, as in Fig. 6. (Early on it was realized that the behavioral range of *one-layer* perceptrons is severely limited; this inhibited for a while the study of perceptrons. It was eventually realized that *multilayer* perceptrons have fully general computing capabilities.) Interest in neural networks remained sparse for 20 years, with occasional contributions from physiologists and physicists. The 1980s saw a sweeping revival, with new ideas from statistical mechanics and dynamical systems, such as *energy function* and *stable attractors* (17); new programming techniques, such as the *back-propagation* learning algorithm (18); and, of course, the availability of computing machinery of ever-increasing performance.

Today, neural networks are used routinely in many specialized applications, chiefly in low-level image and speech processing, and sensors/actuator integration in motor control; they are also widely used for a variety of noncritical tasks where adequate training by example can be imparted rapidly and economically by nonspecialists: data presorting, screening of applications, poll analysis, quality control. On the theoretical side, much of the initiative and of the conceptual machinery for fresh developments has been coming from the statistical-mechanics community. On the architectural side, arguments favoring elements that are simpler, more numerous, and more heavily interconnected than in traditional architectures (see section entitled "Connection Machines") have to vie with the pressure of technological expediency, which favors uniform and local interconnections and limited fanout of signals (see section entitled "Cellular Automata and Lattice Gases").

In the mean time, neural networks have matured enough to provide substantial conceptual and practical contributions to the study of the brain itself. This is the domain of *computational neuroscience.*

Associative Networks

One use for neural networks is pattern classification. Suppose we want to sort a collection of transparencies into "faces," "landscapes," and so on, and possibly "other." To this purpose, we line the two-dimensional projection screen with a collection of neurons like that of Fig. 5; each neuron position defines a *pixel* (picture element). For simplicity, we'll assume that a transparency has only two levels (black and white), so that to each image one can associate a neuron firing pattern (+1 for white and −1 for black), and conversely every firing pattern can be viewed as an image.

The neurons will be interconnected as an *autonomous* network; that is, all neuron inputs come from outputs of other neurons rather than from the outside world. The dynamics is specified by assigning the neuron weights as we shall see in a moment. The initial state of the network is specified by making the neuron firing pattern be a copy of the submitted image. Started from this pattern and left to its own evolution, the network will describe a trajectory through the space of all possible patterns, as indicated in Fig. 7. Each basin of attraction can be thought of as a "concept," and its attractor (which is itself a two-dimensional image) can be thought of as an "exemplar" or "ideogram" for this concept. The network will then behave as an *associative memory:* Confronted with an arbitrary image used as a *key,* it will eventually respond to this key with the corresponding *entry*—that is, the attractor of the basin of attraction in which the key happens to lie. In this way, the classification of points into basins of attraction, which is implicit in the assignment of weights, is made *manifest* by the operation of the network.

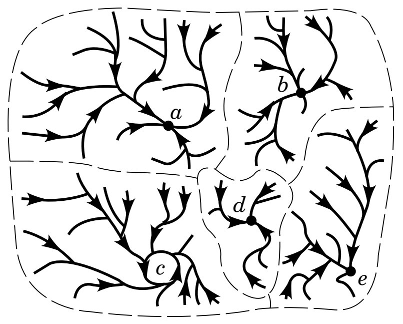

Figure 7. Basins of attraction. Here attractor c is a short cycle rather than a point.

Given specified ideograms $\xi^1, \ldots, \xi^p$, how do we construct a network that will have these ideograms as attractors? In analogy with plausible neurological mechanisms, in the *Hopfield model* (17) the weights are chosen by the Hebb rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} \xi_i^\mu \, \xi_j^\mu \tag{1}$$

where ξ_i^μ denotes the value of ideogram ξ^μ at the i neuron position or pixel, and $w_i j$ denotes the weight with which the output of neuron j enters in neuron i. It turns out that this assignment substantially achieves the goal, provided that the entries are sufficiently distant from one another. The patterns $\xi^1, \ldots, \xi^p$ that define the weights are effectively "stored" in the network, and the evolution will retrieve one of the stored values. In general, the network will have additional attractors besides the specified ones; these are *spurious* entries, and they can be viewed as a way for the network to say "no match" to a key that does not have an obviously matching entry.

A refinement of the above approach, called *simulated annealing* (19), aims to reduce the number of spurious responses. Note that the output from the summation node in Fig. 5 represents the "tendency" for the neuron to fire; however, the neuron will fire if and only if this tendency is above the threshold; The response is all-or-nothing and deterministic, and clearly some of the information available by the neuron is not made use of. Simulated annealing replaces this deterministic response by a stochastic one based on an *energy function* to be minimized (this function is typically derived from the above Hebbian weights) and a *temperature parameter.* This approach has three advantages: (1) While retaining an all-or-nothing firing behavior, one can still grade the neuron's response in a continuous fashion by giving a greater firing *probability* to neurons that would have a greater tendency to fire. (2) The stochastic dynamics corresponds to a *random walk* (with some bias toward lower energies); this makes it possible to backtrack and avoid getting stuck in shallow relative minima. (3) By starting at a high temperature, the search for a significant local minimum is initially coarse and fast; by gradually lowering the temperature, the search becomes slower but more refined; different "annealing" schedules are appropriate for different kinds of problems.

Learning

In the subsection entitled "Associative Networks," the network weights were given. Are there ways to make a network

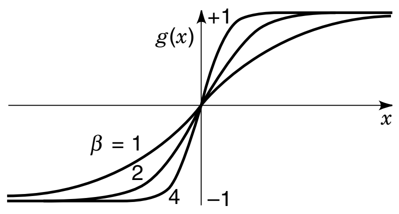

Figure 8. Sigmoid transfer functions are often used in analog networks as an alternative to a step function.

"learn" by itself the weights appropriate for a certain classification? Can we "show" the network a number of pattern templates, and ask the network to figure out the weights that will produce basins having these templates as attractors?

Major progress in this direction was the discovery of the *backpropagation algorithm* (18). Basically, one starts with a perceptron (Fig. 6) and replaces the step function (see Fig. 5) with a continuous, differentiable transfer function having a step slope in the vicinity of μ, such as the *sigmoid:*

$$g(x) = \tanh(\beta x) \equiv \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}} \tag{2}$$

where β is an adjustable parameter (see Fig. 8). In this way, the outputs are differentiable functions of the inputs.

Let x^μ denote an input pattern ($\mu = 1, \ldots, p$), y^μ the corresponding output pattern for a given set of weights, and $\bar{y}^\mu$ the *desired* output pattern for that input. The overall error between actual and desired response will be measured by the following error function (sum of squares):

$$E = \frac{1}{2} \sum_{ij} (y_i^\mu - \bar{y}_i^\mu)^2 \tag{3}$$

It can be shown that, in the present context, E is a differentiable function of the individual neuron weights as well as of the inputs. Proceeding backwards from the outputs, one can adjust the weights one layer at a time so as to minimize the error E at each stage, using the derivatives to determine the direction and rate of correction. This algorithm, which is not very demanding (if n is the number of synapses, one only needs to calculate order n derivatives, while minimization of E by simultaneously adjusting all the weights requires order n^2), is supported both by theoretical considerations and empirical results.

A more ambitious endeavor is *unsupervised learning.* In the training mode, the network is expected to identify and extract significant features of the input stream and build appropriate weights; these weights are then used during the normal mode of operation to classify further input patterns.

CELLULAR AUTOMATA AND LATTICE GASES

Cellular automata are dynamical systems that play in discrete mathematics a role comparable to that partial differential equations in the mathematics of the continuum. In terms of structure as well as applications, they are the computer scientist's counterpart to the physicist's concept of a "field" governed by "field equations." It is not surprising that they have been reinvented innumerable times under different names and within different disciplines; the canonical attribution is to Ulam and von Neuman, circa 1950; much early material is collected in Ref. 20.

In the 13th century, Thomas Aquinas postulated that plants are not reducible to inanimate matter: they need an extra ingredient—a "vegetative soul." To have an animal, you needed a further ingredient—a "sensitive soul." Even that was not enough to make a human; one had to postulate one more ingredient—a "rational soul," William of Occam had replied, Do we really need to put all these souls in our catalog? Might not we be able to make do with less?

An important step toward an answer was taken by Turing in his foundation of logical thought. As we've seen, he showed that, no matter how complex a computation, it can always be reduced to a sequence of elementary operations chosen from a fixed catalog. In this sense, Turing had reduced *thought* to simple, well-understood operations.

Von Neumann was interested in doing for *life* what Turing had done for thought. Conventional models of computation make a distinction between the structural part of a computer (which is fixed) and the data on which the computer operates (which are variable). The computer cannot operate on its own matter; it cannot extend or modify itself, or build other computers. In a *cellular automation,* by contrast, objects that may be interpreted as passive data and objects that may be interpreted as computing devices are both assembled out of the same kind of structural elements and subject to the same fine-grained laws; computation and construction are just two possible modes of activity. Von Neumann was able to show that movement, growth according to a plan, self-reproduction, evolution—*life,* in brief—can be achieved within a cellular automation—a toy world governed by simple discrete rules (21); in that world at least, life is in principle reducible to well-understood mechanisms given *once and for all.* Remarkably, the strategy developed by von Neumann for achieving self-reproduction within a cellular automation is, in its essential lines, the same which a few years later Watson and Crick found being employed by natural genetics.

In a cellular automation, space is represented by a uniform array. To each site of the array, or *cell* (whence the name "cellular"), there is associated a state variable ranging over a finite set—typically just a few bits' worth of data. Time advances in discrete steps, and the dynamics is given by an explicit *rule*—say, a lookup table—through which at every step each cell determines its new state from the current state of its neighbors (Fig. 9). Thus, the system's laws are *local* (no action-at-a-distance) and *uniform* (the same rule applies to all sites); in this respect, they reflect fundamental aspects of physics. Moreover, they are *finitary:* Even though one may be dealing with an indefinitely extended array, the evolution over a finite time of a finite portion of the system can be computed *exactly* by finite means.

The "fire" simulation of Fig. 1 used a cellular automation model. Let each cell have three states, namely, *ready, firing,* and *recovering.* At time $t + 1$, a ready cell will fire with a probability p close to 1 if any of the four adjacent cells (i.e., to the North, South, East, and West) was firing at time t. After firing, the cell will go into the recovering state, from which at each step it has a probability q of returning to the ready state (thus, for small q, the average recovery time is of the order of $1/q$ steps). This yields excitation patterns that spread, die out, and revive much like prairie fires; in this metaphor, p

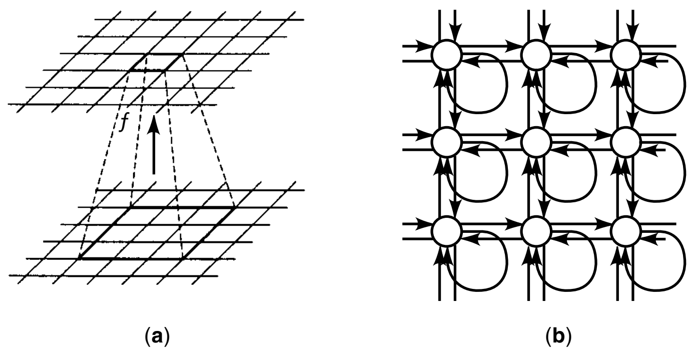

Figure 9. Example of cellular-automaton format: (a) The new state of a cell is computed from the current state of the 3 × 3 block centered on it by the rule *f*, which has nine inputs and one output. (b) Information flow between cells (only vertical and horizontal wires are shown; diagonal ones were suppressed to avoid clutter). Note the feedback loop from each node to itself.

represents the "flammability" and q the "rate of regrowth" of grass (9). Another cellular automaton with a rich phenomenology is Conway's game of "life," which spread as a campus cult in the 1970s (22).

Cellular automata are ideal for modeling the emergence of mesoscopic phenomena when the essence of the microscopic dynamics can be captured by a "board game" of tokens on a mesh (23). This is the case, for example, of *diffusion-limited aggregation* (Fig. 10) and *Ising spin dynamics* (Fig. 11)—a simple model of magnetic materials.

Fluid Dynamics

Experience has shown that in many applications it is more convenient to use, in place of the cellular automaton scheme of Fig. 9, a slightly modified scheme called *lattice gas.* In this scheme, the data are thought of as *signals* that travel from site to site, while the sites themselves represent energy, that is, places where signals interact as in Fig. 12. The lattice-gas scheme was arrived at independently, but in response to similar physical motivations, by a number of researchers (24). It is widely used in fluid dynamics and materials science modeling.

The idea behind *lattice-gas hydrodynamics* is to model a fluid by a system of particles that move in discrete directions at discrete speeds and undergo discrete interactions. In Pomeau's seminal HPP lattice gas, identical particles move at unit speed on a two-dimensional orthogonal lattice, in one of the four possible directions. (Particles are represented by bits; to "move" a particle, you just erase a bit from one lattice site and write a bit in an adjacent site.) Isolated particles move in straight lines. When two particles coming from opposite directions meet, the pair is "annihilated" and a new pair, traveling at right angles to the original one, is "created" [Fig. 13(a)]. In all other cases—that is, when two particles cross one another's paths at right angles [Fig. 13(b)] or when more than two particles meet—all particles just continue straight on their paths.

As soon as the numbers involved become large enough for averages to be meaningful—say, averages over spacetime vol-

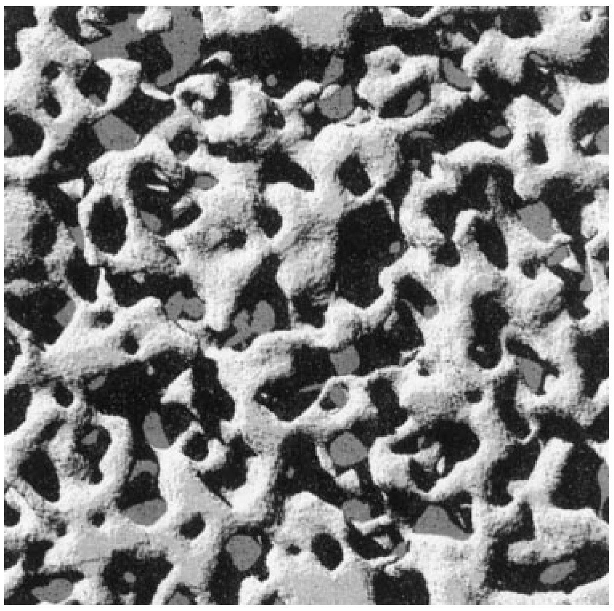

Figure 11. A stage in the cooling of an Ising spin system. Solid matter represents the spin-up phase. Three-dimensional rendering was done by illumination simulated verbatim within the cellular automaton.

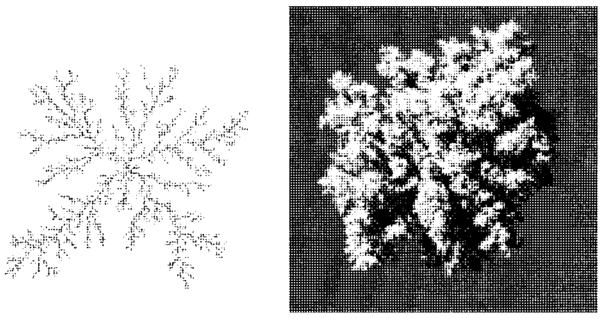

Figure 10. Starting from a nucleation center, dendritic growth is fed by diffusing particles; two- and three-dimensional realizations.

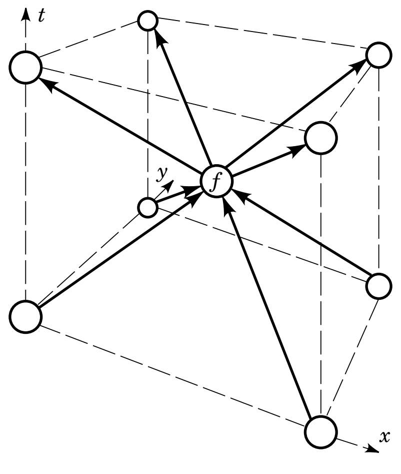

Figure 12. Example of lattice-gas format: Rule *f* has four inputs and four outputs: from the state of the four arcs entering a node (current state), it computes the state of the four arcs leaving the node (new state).

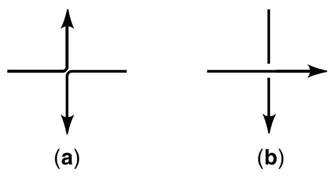

Figure 13. In the HPP gas, particles colliding head-on (a) are scattered at right angles, while particles crossing one another's paths (b) go through unaffected.

ume elements containing thousands of particles and involving thousands of collisions—a definite continuum dynamics emerges. And, in the present example, it is a rudimentary *fluid dynamics*, with quantities recognizable as density, pressure, flow velocity, viscosity, and so on. Figure 14 shows the propagation of a sound wave in the HPP gas. Note that even though individual particles move on an orthogonal lattice, the wave propagates circularly; full rotational invariance has emerged on a macroscopic scale from the mere quarter-turn invariance of the microscopic cellular-automaton rule.

Seeing this fluid model running on an early cellular automata machine (see subsection entitled "Cellular Automata Machines") made Pomeau realize that what had been conceived primarily as a *conceptual* model could indeed be turned, by using suitable hardware, into a *computationally accessible* model; this stimulated interest in finding lattice-gas rules which would provide better models of fluids. A landmark was reached with the slightly more complicated FHP model (it uses six rather than four particle directions), which gives, in an appropriate macroscopic limit, a fluid obeying the well-known Navier–Stokes equation and is thus suitable for modeling actual hydrodynamics (see Ref. 25 for a tutorial). This model started off the burgeoning scientific business of *lattice-gas hydrodynamics.* Soon after, analogous results for three-dimensional models were obtained by a number of researchers (26,27). The approach is able to provide both conceptual (28) and practical insight into more complex situations, such as multiphase fluids and flow in porous media (29), and dynamics that "ride" on the fluid flow, as in Fig. 15 (30).

MOLECULAR COMPUTERS

The smallest electronic devices of today, about 100 nm across, consist of approximately 10^8 atoms; on this scale, a continuum of shapes can still be "machined" and a continuum of compositions "brewed." At the current rate of progress, in 20 years we will reach atomic scale; on this scale, device engineering will have to have made the transition to a different design

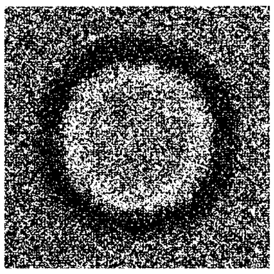

Figure 14. Sound wave propagation in the HPP lattice gas.

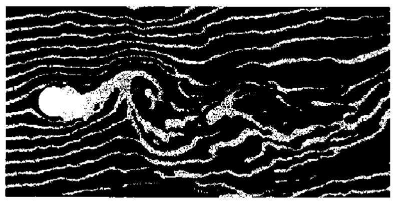

Figure 15. Flow past an obstacle. The tracing is done by injecting into the fluid a "scum" that is dragged by the fluid and whose texture is a compromise between cohesive forces and disruption by shear and thermal agitation. The scum is simulated by a second lattice-gas model, coupled to the first, representing a fluid near the critical condensation point—and thus poised between the gaseous and liquid phases (30).

strategy; namely, devices will have to be assembled from a discrete catalog of parts offered by nature (atoms and electrons), and effects will have to be chosen from the natural interactions between these discrete parts.

The search is on for ways to achieve useful computation in this context. Biochemistry provides a working example. Specifically, DNA (with its RNA variants) is universally used by life as an information-storage medium, and information and materials-processing subroutines are carried out by a standard set of protein-assisted reactions.

DNA Computing

An example of how DNA computing might be domesticated is provided by Adleman's approach (31), which is based on DNA splicing. The computational task he addressed, namely, the *traveling salesman's problem,* is of the following kind. The *domino* game is played with oblong tiles carrying a numerical label (1 through 6) at either end ([1 1], [1 2], . . ., [1 6], [2 1], [2 2], . . .). Tiles can be strung end-to-end, with the constraint that abutting labels match (e.g., [3 4][4 2][2 3]). Let us consider an ensemble of domino pieces satisfying the conditions that (a) all of the labels are represented, (b) some of the possible tiles (e.g., [3 2]) may be missing, and (c) if one tile is present, then it appears in an unlimited number of copies. If one thinks of the labels as "cities," the problem is to determine whether there is a chain that starts and ends with city 1 and passes through all other cities exactly once.

Adleman's technique takes advantage of the fact that, in an appropriate chemical environment, complementary segments of DNA tend to bind together, with the pairing being more stable the longer the extent of the match. In Adleman's experiment, tiles are represented by DNA strings of modest length, namely, 20 DNA bases; the first 10 bases encode the left label (this encoding is unique but otherwise arbitrary), the last 10 encode the right label according to the same code, but using the complementary bases. Thus, if two DNA strings carry labels that match according to the domino rules, then the right-half of one string complements the left-half of the other, and the two strings will tend to splice together. A fresh batch of separate tiles will gradually develop a number of bound complexes, with the great majority of them being legal domino chains. This is a form of *massively parallel* processing; as in a water solution containing 10^{13} tiles, the rate of chain collisions (driven by thermal agitation) may be on the order of 10^{15}/s.

Besides this step that spontaneously generates random legal chains, the procedure employs other steps (always carried out by massively parallel chemical reactions), which help to efficiently steer the search toward the problem's solution. Specifically, one uses techniques for (a) amplifying the number of partial chains which meet the problem's requirements and (b) weeding out those that don't. If at the end of this procedure there are any chains left, these represent solutions of the problem; otherwise, the problem has no solutions.

Thus, Adleman's technique can solve the traveling salesman's problem for a small number of cities. Since this problem is NP-complete (this term denotes a well-characterized "degree of intractability") and NP-complete problems are widely believed (though not quite proved) to be of exponential complexity, speculation has arisen that life processes of this kind could carry out tasks transcending the capabilities of conventional computers. The present approach provides no support for this thesis. In fact, though the number of steps in the procedure increases only linearly with the number of cities, the number of DNA molecules in a batch must grow exponentially. In the end, the physical tradeoffs are of the same general nature as with other parallel schemes.

Molecular Nanotechnology

A number of activities related to molecular nanotechnology have found a rallying point in the Foresight Institute (32) Drexler's manifesto (33) places specific emphasis on computational issues. In this sense, however, "nanotechnology" does not represent so much a well-defined discipline as a clearing house for a miscellanea of initiatives aimed at harnessing atomic-scale mechanisms to computation and fabrication goals.

Cellular Computers

DNA computing as previously discussed borrows techniques and materials from biochemistry; the "program," however is a sequence of externally driven, in vitro reactions that are set up in a conventional laboratory by conventional macroscopic means, much like photofinishing. We already know how to induce bacteria to synthesize in commercial quantities "designer's chemicals" specified by us. Can we program a cell to carry out in vivo, within the cell itself, a sequence of microscopic biochemical steps that correspond to recognizable logic operations?

This is an ambitious but not implausible undertaking. The approach suggested by Knight and Sussman (33a), for one, builds entirely on existing cellular mechanisms. A logic variable is represented by the concentration (low versus high) of a particular protein; different variables are encoded in different proteins. The idea is to use DNA-binding proteins, so that the protein that expresses one gene acts, according to the case, as a repressor or a promoter for the expression of another gene. This feedback loop provides the three basic effects—amplification, inversion, and gating—by which digital circuitry can be made to emerge from analog mechanisms.

SWARM COMPUTERS

Phenomena involving diffusion and reactions of molecules are well known in chemistry. When the entities involved are substantially more complex than molecules, such as small self-propelled animals or artifacts, one speaks of *swarm computation.* The study of the possibilities of this mode of computation is still in its infancy. We can't do better than refer the reader to Ref. 7 for a popular but well-documented reportage on this field.

SOME ACTUAL MACHINES

Connection Machines

Connection Machines originated at the MIT Artificial Intelligence Laboratory, and they reflect a tradition of artificial intelligence (AI) problems and LISP programming environment. They were the standard bearers of "connectionism"; this is a computing philosophy that stresses (a) the use a large number of small processors and (b) giving the interconnection pattern as much importance as the instruction stream as a means to program the structure for a particular task.

In its original formulation (58), the connection machine was intended to be an efficient digital-hardware platform for computations requiring fine grain and flexible connectivity. Each element would communicate with any other by broadcasting in a spherical wavefront a packet of information together with the destination address, and it would be the responsibility of the recipient to recognize the address and intercept the packet. Eventually, for practical reasons, the architecture evolved into something more like a cellular automation, with two important differences: (1) The rule table was sequentially broadcast from an external host, and thus could be changed from step to step under host program control; and (2) in addition to the cellular automaton's hard-wired local-and-uniform interconnection pattern, a higher level of interconnection (point-to-point and software-handled) was provided by a programmable router (34).

The embarassing lack of enthusiasm with which the AI community received the first connection machine (CM-1) has been adduced as evidence that this architecture did not, after all, provide what AI had requested. More likely, the connection machine was what AI people claimed they wanted; but in fact it called their bluff, because the AI community was not ready yet to actually make full use a connectionistic architecture.

As an afterthought, a small number of high-performance floating-point processors had been interspersed through the fine-grained array of the CM-1. These proved to be very useful in a number of mundane problems like image processing and lattice-gas hydrodynamics (see below). Instead of performing an ancillary function, the floating-point processors came to the forefront, and the underlying fine-grained texture was more often than not used as a programmable "conveyor belt" to feed these processors. This reality was reflected in the CM-2 design, which for a time held its own among "scientific" (i.e., number crunching) supercomputers.

Eventually the design evolved into an original but somewhat more conventional architecture, the CM-5 (35), consisting of a cluster of RISC processors (of the Sparc type) connected by a fat-tree (36) network operating in packet-switching mode. (This is a *fractal* network structure, and it represents an alternative way to embed a few levels of exponential growth within polynomial spacetime.) Subsequent onslaught by commodity microprocessors and affordable, fast local-area networks gradually robbed this architecture of much of its competitiveness.

Cellular Automata Machines

We refer here to a lineage of machines that provide, rather than a specific cellular automaton, machinery for efficiently synthesizing a variety of cellular automata architectures in any reasonable number of dimensions. This approach, which combines flexibility with efficiency, has been termed "programmable matter" (37).

With current technology, one can build a memory chip holding 64 Mbits for an indefinite amount of time at virtually zero dissipation (just occasional refreshing) and allowing one to access bits at a gigahertz rate with a dissipation of about 1 W. With the same technology, one could build a simple cellular-automaton cell on a 20 μm square and put 1 K $\times$ 1 K cells on a chip. Since in this architecture each driver would see a small fixed load at a small fixed distance, cells could in principle be clocked at microwave rate (say, 10 GHz) for a total of 10^{16} events/s. However, the whole chip would then dissipate *thousands of watts!*

For sake of comparison, let's note that chips remarkably similar to a cellular automaton are actually being made today. These are field-programmable gate arrays (FPGAs), consisting of a regular array of macrocells (each having a few bits of storage for state-variables, a lookup table for the dynamics, and assorted routing circuitry). However, these cells are meant to be sparsely interconnected on a *chip-wide* scale; the attendant propagation delays limit clocking rate to about 100 MHz, and at this rate the largest such chips dissipate a few watts. That is, in an FPGA the event rate may be hundreds of times lower than that of a cellular-automaton array, and correspondingly the dissipation hundreds of times smaller.

In sum, our capabilities to compute large numbers of events are limited not so much by how many cells we can squeeze in a chip or by how fast we can clock them, as by *how much energy an event dissipates!* It is true that, as technology steadily progresses from "submicron" to "nanoscale," the dissipation per event is likely to decrease. But devices will be smaller and faster, and according to current scaling trends the dissipation *per unit area* is likely to *increase!*

Thus, it may be preferable to optimize the event processor, where most of the dissipation lies, and multiplex it between many memory sites. Accordingly, an earlier cellular automata machine designed at the MIT Laboratory for Computer Science (23) time-shared a single processor between hundreds of thousands of cells. The rule processor for CAM-PC was simply a lookup table, consisting of a fast SRAM (static RAM); the cells were stored in a DRAM (dynamic RAM) chip. With a minimum of glue logic to shuttle data between SRAM and DRAM, and using the access pattern most natural to the DRAM memory, both SRAM and DRAM were used at full bandwidth. Since these are commodity chips, the combination was very cost-effective; however, the cell interconnection pattern was essentially given once and for all.

The CAM-8 design (37) allows one to seamlessly integrate an indefinite number of modules of this kind, each consisting of a SRAM processor shared between millions of DRAM cells, and at the same time achieve, under software control, any desired cellular-automaton interconnection pattern, without restricting access to just first neighbors.

Physically, CAM-8 is a three-dimensional mesh of *modules* (a module is akin to a frame buffer with on-board processing resources) operating in lockstep on pipelined data. This structure is dedicated to supporting a variety of *virtual* architectures in which massively parallel, fine-grained computation takes place, using the lattice-gas scheme, on a mesh that may consist of billions of sites. The virtualization ratio—that is, the ratio between the number of virtual processors and that of real processors—may be set from hundreds to millions.

To visualize the operation of CAM-8, consider a regular n-dimensional array of bits that extends indefinitely in all directions (for concreteness, one may think of a two-dimensional array—a "bit-plane"); we shall call such an array a *layer.* We shall now superpose, in good registration, a number p of layers—so that at each site we have a *pile* of p bits. This entire collection of bits will be made to evolve by repeated application of the following procedure, called a *step,* consisting of two stages:

- *Data Convection.* Each layer is independently shifted as a whole by an arbitrary number of positions in an arbitrary direction. We still end up with a pile at each site, but with a new makeup.
- *Data Interaction.* We now take each pile and send it to a p-input, p-output lookup table; this table returns a new pile, which we put in place of the original one.

Note that at the data interaction stage each pile is processed independently, so that the *order* in which the piles are updated is irrelevant. One could even have several copies of the lookup table and do some (or all) of the processing concurrently. In CAM-8, the mesh is apportioned between the modules; each module works serially on its portion, and all the modules operate in parallel.

Also note that at the data convection stage, the shift performed on each layer is a uniform and data-blind operation (each bit is moved by a fixed offset, independently of its address and value). Thus, in a suitable implementation, it becomes possible to replace this operation by one that shifts the frame of reference (by incrementing a single pointer) rather than moving the data themselves. This is indeed the case in CAM-8, where, within a module, each layer is scanned serially by a set of nested DO loops with each nesting level corresponding to one spatial dimension. By adding an offset to the loop index of a given layer, one shifts by the same amount the order of access of sites within that layer. The entire layer then will be accessed in the same order as if the data themselves had been shifted. Near the edges of a module, an address within the module may, after the offset, actually point to data outside the module. A lockstep data-passing arrangement ensures that data are brought in as required from adjacent modules in a seamless fashion.

To sum up, CAM-8 realizes a cellular automata architecture in which the following features (besides the rule table itself) are programmable:

- The global geometry of the virtual mesh: the number of dimensions, the length along each dimension.
- The number of lattice-gas signals involved at each site, and the number of bits for each signal.
- The interconnection between sites and the interaction of

data at a site. Interconnection and interaction may be reassigned from step to step. This allows one to realize time-dependent dynamics; it also allows one to synthesize complex interaction "macros" as sequences of simpler interactions.
- The virtualization ratio, as mentioned above.

Machines like CAM-8 address an almost unexplored band of the computational spectrum and rely on a different programming approach than conventional computers. It is true that on naturally fitting tasks they may yield a performance gain of two to three orders of magnitude; however, this gain is to a large extent offset by the economies of scale, in hardware and software, enjoyed by the mass computer market.

CONSERVATIVE LOGIC

All computers, including electronic and biological ones, run, of course, on physics. However, the essential point of computation is that the physics is segregated once and for all *within* the logic primitives (say, gates and wires). Once one is given the formal specifications of these primitives (such as the input/output table for the NAND gate, as in Fig. 2) and perhaps some design constraints (time delay through a gate, speed of propagation along a wire, maximum number of inputs that an output can drive), one can forget about the physics that is behind the logic: programming is an exercise in virtual reality, not in physics (38).

Precisely because logic isolates one from physics, the only physical resources that one can manage at the programming level are those that are indirectly reflected in the logic; thus, though one cannot double the amount of physically available RAM by clever programming, one might still be able to achieve an equivalent result by running a data compression algorithm.

Here we shall discuss attempts to incorporate more aspects of physics into the formal scheme of computation, giving the programmer greater scope for physical resource management *from within the logic itself.* One aim is to achieve a better match between the logic of a program and the underlying physics, and thus, ultimately, better performance. As a bonus, one gains a better understanding of the "information mechanical" aspects of physics.

Three Sources of Dissipation

In this section we address what are basically thermodynamical aspects of computation (39).

A *magnetic bubble* is a small magnetic domain pointing opposite to the surrounding material (see Fig. 16). In bubble memories (40), the two states (1 and 0) of a bit are represented by the presence or absence of a bubble at a given place. By suitable sequencing of external magnetic fields, a row of bubbles can be made to advance along a preassigned path and, in particular, to stream past a reading head much like magnetic tape. Since bubbles do feel the influence of nearby bubbles, it is conceivable that one could use bubbles for logic as well as for storage. Note that conventional logic elements (see Fig. 2) do not preserve the number of 1s (for example, NOT turns a 1 into a 0 and vice versa), and thus would have to contain bubble "factories" and bubble "dumps." On the other hand, though easy to move, bubbles are hard to create and destroy. Is it possible to do general computation by mechanisms that just steer bubbles (41)? A similar problem arises in ordinary CMOS logic, where 1 and 0 are represented by the presence or absence of charge in a capacitor. In conventional CMOS circuitry, a 1 is created by charging a capacitor from a constant-voltage source (the power supply) via a resistor, and it is destroyed by discharging the capacitor to ground, always via a resistor. In either case, the charge transfer that converts a 0 token into a 1 token or vice versa is accompanied by energy dissipation. Thus, for the circuit to operate we must keep supplying high-grade energy and removing heat.

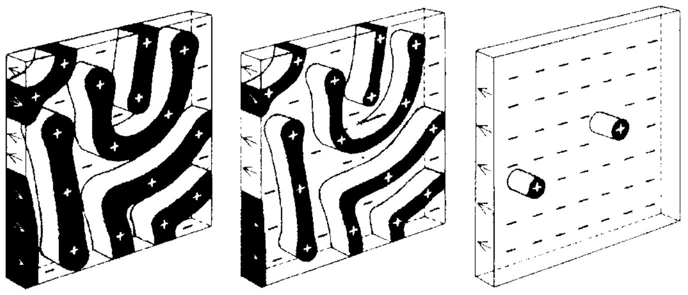
Figure 16. In a suitable two-dimensional magnetic material, serpentine domains of alternating magnetic orientation are spontaneously formed (left). As an increasing external field is applied, domains whose polarity oppose that of the field shrink (middle), until only small cylindrical configurations, or bubbles, remain (right). (Adapted from Ref. 40.)

A more subtle source of dissipation was pointed out by Landauer (42). In the AND gate, three of the four possible input configurations, namely 00, 01, and 11, yield the same result. In this sense, the gate is not *logically reversible* (by contrast, the NOT element is reversible). But, at a microscopic level, physics is presumed to be strictly reversible (this applies both to classical and quantum physics). Thus, the degrees of freedom represented by the logic values can only be a partial description of the physics; to retain reversibility, for every "merge" of trajectories at the logic level there must be a "split" of trajectories in some other degrees of freedom of the system [this is just another way of expressing the second principle of thermodynamics (39)]. No matter how clever we might be in circumventing other sources of energy dissipation, the fact remains that any erasure of information from the logic degrees of freedom of the system must be matched by a proportional increase of entropy in the rest of the system.

Finally, in ordinary computers (just as in brains), signals are continually *regenerated.* Signal regeneration encompasses of a number of housekeeping functions such as noise abatement and signal amplification, and it is really a form of *erasure.* What is thrown out in this case is not logic data (as in clearing a register, when both 0 and 1 are forced to 0), but whatever deviations may have crept into the logic levels because of undesired disturbances; thus, anything that happens to be near a value of 1 (and so is presumably a slightly corrupted version of a logic 1) is forced to 1, and anything near 0 is forced to 0.

CONSERVATIVE-LOGIC GATES

The above dissipative processes—*token conversion, entropy balance,* and *signal regeneration*—are ancillary to a computer's primary business, which is *token interaction.* However, in conventional computers (just as in brains) these ancillary functions are all *bundled* together in the mechanism of a logic element. By unbundling them, *conservative logic* gives one the freedom to handle them separately and to recombine them (possibly at the circuit level rather than at the gate level) so as to better satisfy specific constraints and fulfill specific optimization goals.

For the sake of illustration we shall compare an ordinary logic gate such as the NAND gate with a conservative-logic gate such as the *Fredkin* gate, which, unlike commonly used gates, is *invertible* and *token-conserving.* In ordinary logic, it is assumed that *fanout* is available—that is, that the same output signal can be fed as an input to more than one gate. With this understanding, as already mentioned, the NAND gate (Fig. 17, left) is a universal logic primitive.

In conservative logic, signal fanout as such is not used (signal copies are made by means of gates, not by tapping a wire); on the other hand, certain computations require constant inputs in addition to the argument, and they produce unrequested (or "garbage") outputs in addition to the result. With this understanding, also the Fredkin gate (Fig. 17, right) is universal (43). In fact there are simple transliteration rules for constructing, from an arbitrary logic circuit, a functionally equivalent conservative-logic circuit. Figure 18 shows how to realize some common logic functions.

The NAND gate has two inputs and one output. As shown in the bottom panel of Fig. 17, the entire energy of the incoming signals is ultimately dumped into the heat sink, no matter how much or how little noise might have managed to creep into the signals themselves. The energy of the output signal comes from the power supply; when the output drives more than one load (in the figure, a fanout of 2 is indicated) it will draw from the power supply a proportionate amount of energy.

The Fredkin gate (Fig. 17, right) has three inputs and three outputs. The first signal, u, always goes through unchanged, while the other two come out either straight or swapped depending on whether u equals 1 or 0. Thus, here

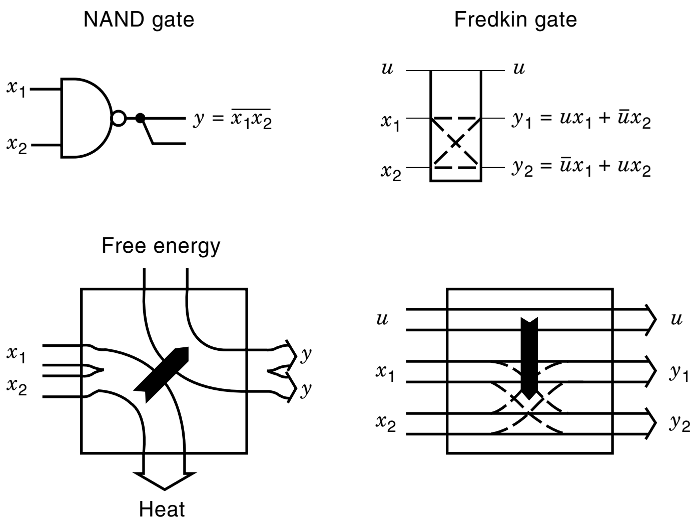

Figure 17. Logic diagram (top) and energy flow (bottom) of the NAND gate and the Fredkin gate. The NAND gate is shown with a fanout of 2. The solid arrows indicate the relevant interactions.

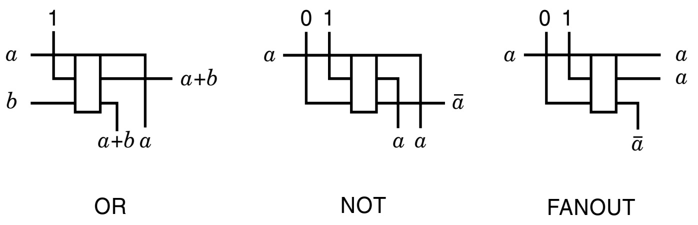

Figure 18. Realization of the OR, NOT, and FANOUT functions by means of the Fredkin gate. Inputs are from the left, outputs to the right. The quantities (0s and 1s) that flow in from the top are *constants;* those that flow out from the bottom are garbage, to be recycled. For instance, in the left panel, inputs a and b yield as a result their logical OR, denoted by $a + b$; an input constant of 1 is needed for the Fredkin gate to operate as desired, and two garbage values, $\bar{a} + b$ and a, are produced.

the entire energy of the output signals comes from the input signals. (If one suspects that a signal may have become attenuated or contaminated by noise, it will be one's responsibility to pass it through a "restoring bath" of strength commensurate to the expected amount of degradation; it is only there that free energy will be drawn from the power supply.) Note that only "single strength" signals are provided at the gate's output; it is the circuit designer's responsibility to insert additional gates to perform fanout if and when copies are required. In this way, copies are paid for only when needed. And once one is done with a signal, in most cases conservative logic provides the means to recycle the energy temporarily invested in it (43) (Fig. 23). Logic recycling in reversible computation was introduced by Bennett (44); more details can be found in Ref. 43.

To summarize, conservative logic is a scheme for computation based on discrete operations (*events*) on discrete objects (*signals*), and this scheme satisfies three independent *conservation laws,* namely:

- *Conservation of the Number of Threads.* Each event has as many output signals as input signals, and composition of events matches outputs to inputs on a one-to-one basis; thus, a computation can be thought of as the time evolution of a fixed collection of binary degrees of freedom, or *threads.* The state of each thread may change from event to event, but *the number of threads is invariant.*
- *Conservation of the Number of Tokens.* Let logic 1 and 0 be represented by two kinds of token (or, equivalently, by the *presence* or *absence* of a token at a given place). At each event, the tokens carried by the threads that participate in that event may be reshuffled, but *the number of tokens of each kind is invariant.*
- *Conservation of Information.* Finally, conservative logic is invertible; that is, each event establishes a one-to-one correspondence between the collective state of its input signals and that of its output signals. As a consequence, the current global state of the system uniquely determines the system's entire past as well as its future. If our knowledge of the initial state of the system is expressed by a statistical distribution, then this distribution will in general change as the computation progresses, but *its entropy is invariant.*

A conservative-logic computation may be visualized as a piece of spacetime tapestry, with threads running in time's general direction. At each point in spacetime the threads are liable to cross or change color, but the flow of material, color, and information obeys a strict accounting discipline much like that imposed on an electric circuit by Kirchhoff's laws.

Replacing conventional logic elements with conservative-logic ones eliminates two of the sources of dissipation listed at the beginning of this section, namely, token creation/destruction (or token conversion) and logically irreversible operations; moreover, it relieves the individual gate of the responsibility for signal regeneration, so that the latter can be performed when and where needed rather than at every step. All this would be of no avail if conservative-logic elements were not *concretely realizable.* In the next two sections we'll illustrate two physical implementations of the conservative-logic scheme, and at the end we'll discuss some of the *costs* of this approach.

A Billiard-Ball Computer

As we have mentioned, the energetics of magnetic bubbles puts a premium on circuit design principles that help conserve bubbles. Ideally, *logic interaction* of tokens should reduce to mere *course deflection.* A general way to achieve this goal was indicated by the well-known *billiard-ball model of computation* (43). [There were other computational schemes, invented merely to conserve tokens (45) which do not share conservative logic's additional concern for thread and entropy conservation.]

In the billiard-ball model the primitives of conservative logic are realized by elastic collisions involving *balls* and fixed *reflectors.* Note that the "rules of the game" are identical to those of the idealized physics that underlies the classical theory of ideal gases (where the balls represent gas molecules and the reflectors represent the container's walls). Quite literally, just by giving the container a suitable shape (which corresponds to the computer's *hardware*) and giving the balls suitable initial conditions (which correspond to the *software*—program and input data), one can carry out any specified computation.

In this scheme, the nonlinear effect which provides the computing capabilities is simply the collisions of two balls, as indicated in Fig. 19(a). Note that a ball will emerge at the upper output, labeled pq, if balls are present at both inputs ("p AND q"), while one will appear at the output below it ($\bar{p}q$) if the ball on the upper input is absent ("NOT p AND q"). The role of wires is served by hard mirrors, which "focus" balls back into the fray. Figure 19(b) shows a *switch gate* (invented independently by Ed Fredkin and Richard Feynman). One Fredkin gate can be constructed out of four switch gates and a few additional mirrors (43).

Thus, general computation can be achieved without creating or destroying balls; all one needs is *conditional permutations* of balls, as prescribed by the Fredkin gate (Fig. 18). In turn, the required permutations may be synthesized from simple two-particle interactions of the kind contemplated in elementary mechanics. See the end of the next section for a critique.

A Charge-Permuting Computer

Here we present a realization of conservative-logic in which the tokens to be processed are unit *charges* instead of bubbles

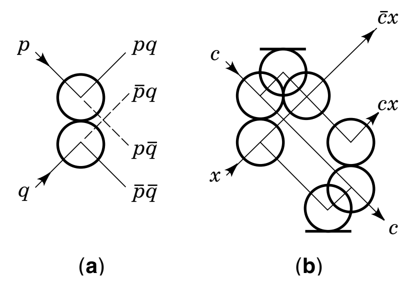

Figure 19. (a) The basic nonlinear effect of the billiard-ball scheme, namely, the collision of two hard spheres of finite diameter. The labels are logical expression whose values are the presence or absence of a ball on the corresponding path; thus p is true if a ball is injected at upper left. The label pq indicates that a ball will emerge from, say, the upper output only if both input balls are present ("p AND q"). (b) A billiard-ball realization of the *switch gate* (the Fredkin gate may be built out of four of these). The thick lines indicate mirrors; the circles indicate snapshots of balls taken at collision instants. If there is no ball at the "control input" c, a ball at x will go through undeflected and come out at $\bar{c}x$; if a ball is present at c, a ball at x will collide with it, the two balls will exchange roles, and eventually a ball will come out at cx.

or balls. This scheme, introduced by Fredkin and the author 30 years ago (46), is the conceptual forefather of a family of technological approaches that has started blossoming in the last few years in connection with low-power computing strategies (see subsection entitled "Adiabatic Charge Recycling").

As we've seen, conservative logic is thread-conserving. Thus, a circuit can be drawn as a collection of threads running parallel to one another (a thread can be visualized as a shift register), with Fredkin gates conditionally swapping data between pairs of threads, as in Fig. 20.

In turn, a Fredkin gate is realized as a two-pole, double-throw switch. With complementary metal oxide semiconductor (CMOS) technology, it is possible to make almost ideal switches that require no power to hold the switch on or off (the control electrode responds like a capacitor C in series with a small resistor R_c); the switch itself has a virtually infinite "off" resistance and a small "on" resistance R_s. In the case we are going to discuss, a control electrode is always driven by a switch, so that the only relevant resistance is the series combination $R = R_c + R_s$.

For a moment we'll ignore this resistance ($R = 0$), but we will explicitly represent the capacitor C. To avoid an infinite inrush current a capacitor must be charged and discharged via an inductor L. With these provisions, circuitry like that of Fig. 20 will look like Fig. 21. The starred capacitors are those associated with the switches' control electrodes; the other, matching capacitors have been added in order to equalize de-

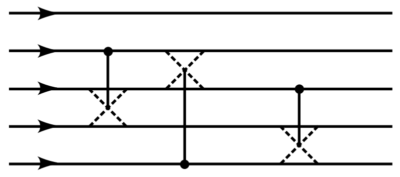

Figure 20. Here a conservative-logic circuit is viewed as a collection of shift registers running parallel to one another. Through a Fredkin gate, the datum in one *control line* determines whether the data in two *controlled lines* are swapped or go straight through.

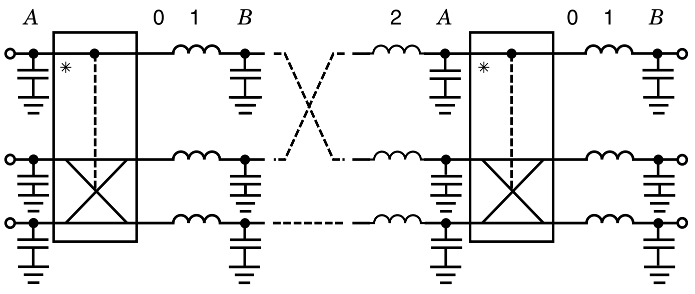

Figure 21. The threads of Fig. 20 are here realized as transmission lines with occasional crossovers. Active stages (A to B) alternate with passive ones (B to A). Logic is done by conditional crossovers (Fredkin gates), and takes place at active stages, while signal routing is done by hard-wired crossovers between threads, and takes place at passive stages. The flow of charges across a stage is timed by semaphore switches, not indicated here but detailed in Fig. 22.

lays on all threads. What we have is a collection of transmission lines with occasional crossovers between lines—some conditional (logic) and some hard-wired (wiring). The flow of charges across a stage is timed by semaphore switches, detailed in Fig. 22. These switches are activated so that the Fredkin-gate data are moved across inductors 0 first, while the control charge remains at A; once the data have been transferred, then the control charge itself is transferred across inductor 1.

In a lumped transmission line, like these, charges will tend to spread as they travel. Since we want to keep the charges localized, as they represent discrete logic tokens, additional switches will be added to regulate charge movement; unlike the Fredkin-gate switches, these will be controlled from the outside and operated according to a fixed schedule in a *data-blind* way—like a traffic light. This arrangement is detailed in Fig. 22.

Both the billiard-ball scheme and the charge-permuting scheme, as discussed so far, are somewhat idealized, and a brief critique is in order. One can identify three basic sources of error:

1. Because of unavoidable fabrication and operating errors (mirror positioning, initial ball position and velocity, thermal noise), the overall trajectory will gradually drift away from the nominal course.

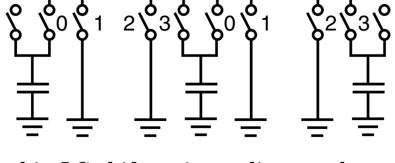

Figure 22. In this LC shift register, discrete charges representing bits hop from capacitor to capacitor through inductors. Charge movement is regulated by a four-phase switching sequence. Starting the cycle when all switches are open and charges are at rest in the capacitors, (a) close switches 0 and 2. Current builds up in the inductors. (b) When the capacitors are discharged and current is at peak, close 1 and open 0, isolating the capacitors. Now current recirculates in the inductors. (c) Open switch 2 and close 3. Energy flows rightwards from inductors into capacitors. (d) When capacitors are fully charged and current is zero, open 1 and 3, completing the cycle.

2. Because of unavoidable *friction,* balls will gradually slow down.
3. When a ball is hit hard, some of the impact energy is spilled onto the ball's internal oscillation modes. This has two consequences: (a) The ball will exit the collision with less than the nominal unit speed, and (b) the next collision will be disturbed by this internal oscillation in a *practically* unpredictable way. (This source of error could in principle be predicted and corrected; but to do so would require additional computing machinery of the same kind as that which we are trying to correct, and the latter would have to be corrected in turn.)

Analogous considerations apply to the charge-permuting scheme, where, for instance, friction is replaced by the ohmic loss when a current encounters a nonzero resistance R.

All three error sources mandate the occasional insertion of a signal-regeneration stage, with attendant energy requirements. The larger the error, the more often one will have to compensate for it by regeneration. Error source 1 can be reduced by better control of fabrication tolerances and environmental disturbances. As a rule, sources 2 and 3 can be reduced by just operating the entire computer more slowly (typically, friction is proportional to the square of velocity and ohmic loss to the square of the current). From a more detailed analysis, one can generally conclude the following:

- There is no fixed amount of energy dictated by physics that one *must* spend for a given computational task. Rather, if one uses a conservative-logic scheme, the same task can be accomplished with less and less *overall* energy expenditure, at the cost of having to wait a proportionally longer time for the result.
- A conservative-logic scheme requires more circuitry than a conventional scheme. Intuitively, one has to complement the computational infrastructure with a whole *recycling* infrastructure. Even though the latter may help one save on operating costs (energy), it requires an additional investment in capital (gates and wires) and real estate (chip area). The overall benefit depends on the relative cost of these resources.

Under certain reasonable assumptions (such as bounded density of waste heat flow), it can be proved that the benefits of conservative/reversible recycling grow faster (asymptotically, by a polynomial factor in the problem size) than its costs (46a).

Adiabatic Charge Recycling

There are a growing number of experimental circuit designs that apply conservative-logic concepts to the goal of lowering the power needs of computers (47–49), and they can all be usefully viewed as variants of the charge-permuting scheme of the section entitled "A Charge-Permuting Computer." Most of these designs are not concerned with the reversibility aspect of conservative logic; in fact, today the energy dissipation due to logic irreversibility is still many orders of magnitude less that that due to what we have called token conversion.

While near-ideal capacitors are easy to incorporate on a silicon chip, inductors tend to be large and lossy. Instead of using an inductor to move a charge across a large voltage gap with little dissipation, *adiabatic charge recycling* achieves a similar result by (1) using a ladder of graded voltage levels and (2) only transferring charges (through the switch resistance R) between adjacent levels. Intuitively, one may think of the power supply as giant LC "flywheel" external to the chip, whose voltage oscillates on a regular cycle. To bring a charge from a point P_1 at voltage V_1 to a point P_2 at voltage V_2, one waits until the flywheel reaches a value close to V_1, connects P_1 to it by a switch, and transfers the charge to the power supply. When the flywheel reaches a voltage close to V_2, the charge is transferred in a similar way to P_2. Thus, by means of these "multiplexing" switches, a large number of small on-chip inductors is replaced by a single, large off-chip inductor.

QUANTUM COMPUTATION

As we've seen, there are aspects of physics, such as reversibility, that are relevant to computation and can be brought under better control by incorporating them directly into the computation scheme. *Quantum computation* represents an important further step in this direction. Quantum mechanics is of course used extensively in the design of semiconductor devices and communication systems. However, until recently the most peculiar, nonclassical aspect of quantum mechanics were hidden within the devices and didn't affect the logic variables that are the object of a computation. In quantum computation the collection of these variables is encoded in a quantum state, and a computation step is the result of a unitary evolution operator acting on this state. Effects such as quantum superposition and interference of different computational states, entanglement between different parts of the system, and so on, are part and parcel of the computation process itself and can be directly controlled and exploited by a program.

It must be noted that adding quantum effects to one's computational tool kit does not make computable any functions that were formerly uncomputable; however, it may make tractable some functions that were formerly untractable. Specifically, while the factoring of integers is a task of exponential difficulty for the best of today's algorithms, Shor recently showed (50) that, in principle, factoring can be done in polynomial time by a quantum computer.

One novel aspect of quantum-mechanical information handling is easy to illustrate. A basic fact of quantum mechanics is that an unknown quantum state cannot be *cloned.* Thus, if information is encoded in a quantum state and transmitted over an insecure channel, it is impossible for a third party to acquire part of this information without giving the sender/receiver team evidence that the channel has been tapped.

Since simulating a quantum system by a classical computer requires an effort exponential in the size of the system itself, Feynman (51), had suggested simulating quantum systems in polynomial time by computers that could avail themselves of quantum resources (intuitively, quantum "opcodes" in addition to conventional ones); a general solution to this problem was soon found by Deutsch (57). Another paper by Feynman (52) expressed the consensus that computers based wholly on quantum mechanics could do conventional digital computation. Soon ways were found to use such computing schemes in unconventional ways, showing the existence of functions whose evaluation could be speeded up by quantum methods. The first functions found in this way were of purely academic interest, but they were followed by Shor's result on factoring, which is a problem of great practical interest in cryptography. At the same time, the advantage of quantum methods for secure communication were being explored by Bennett, Brassard, and others. Quantum teleportation is a theme of much appeal. Quantum logic primitives and circuit design techniques have now reached a certain degree of maturity (53). See Ref. 54 for an introductory article, Ref. 55 for an overall review and references, and Ref. 56 for recent proceedings.

Today the field is still in rapid expansion; and experimental realizations of rudimentary quantum computers, involving a few bits and a few gates, abound. One important concern is *error correction,* which in a quantum context is much more taxing than in ordinary digital logic. Another concern is the investment in ancillary physical resources (fabrication tolerances, shielding, energy dissipation, etc.) that are needed to retain quantum coherence over an increasing number of bits and clock cycles: How fast does this investment scale with the size of the quantum system? Even as quantum mechanics empowers computation, tasks of a computational nature help us probe the power, the limits, and the very meaning of quantum mechanics.

CONCLUSIONS

To protoneolithic man, farming must have seemed a marginal and pretty unconventional way to make a living compared to mammoth hunting. Many a computing scheme that today is viewed as unconventional may well be so because its time hasn't come yet—or is already gone. Some will challenge our ingenuity; at the very least, they are all part of our intellectual history.

BIBLIOGRAPHY

1a. C. S. Calude, J. Casti, and M. Dinneen (eds.), Unconventional models of computation, *Proc. UMC98, First International Conf. Unconv. Models Computation,* Auckland, NZ, 1998, Springer-Verlag, 1998.

1. M. Minsky, *Computation: Finite and Infinite Machines,* Englewood Cliffs, NJ: Prentice-Hall, 1967.
2. C. Mead, *Analog VLSI and Neural Systems,* Reading, MA: Addison-Wesley, 1989.
3. D. Gajski and J.-K. Peir, Essential issues in multiprocessor systems, *Computer,* **18** (6): 9–27, 1985.
4. L. Haynes et al., A survey of highly parallel computing, *Computer,* **15** (1): 7–8, 1982.
5. J. Villasenor and W. Mangione-Smith, Configurable computing, *Sci. Amer.,* **276** (6): 66–71, 1997.
6. E. A. Jackson, *Perspectives of Nonlinear Dynamics,* Cambridge, UK: Cambridge Univ. Press, 1991.
7. K. Kelly, *Out of Control: The New Biology of Machines, Social Systems and the Economic World,* Reading, MA: Addison-Wesley, 1995.
8. D. Ruelle, *Chance and Chaos,* Princeton, NJ: Princeton Univ. Press, 1991.
9. M. Schroeder, *Fractals, Chaos, Power Laws,* New York: Freeman, 1991.

10. H. Abelson and G. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs,* Cambridge, MA: MIT Press, 1985.
11. T.-Y. Feng, A survey of interconnection networks, *Computer,* **14** (12): 12–27, 1981.
12. B. Hayes, Collective wisdom, *Amer. Sci.,* **86**: 118–122, 1998.
13. M. Flynn, Very high speed computing systems, *Proc. IEEE,* **54**: 1901–1909, 1996.
14. J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation,* Reading, MA: Addison-Wesley, 1991.
15. W. S. McCulloch and W. Pitts, A logical calculus of ideas immanent in nervous activity, *Bull. Math. Biophys.,* **5**: 115–133, 1943.
16. F. Rosenblatt, *Principles of Neurodynamics,* New York: Spartan, 1962.
17. J. J. Hopfield, Neural networks and physical systems with emergent collective computational abilities, *Proc. Natl. Acad. Sci. U.S.A.,* **79**: 2554–2558, 1982.
18. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, *Nature,* **323**: 533–536, 1986.
19. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, Optimization by simulated annealing, *Science,* **220**: 671–680, 1983.
20. A. Burks, *Essays on Cellular Automata,* Chicago: Univ. Illinois Press, 1970.
21. J. von Neumann, *Theory of Self-Reproducing Automata* (edited and completed by Arthur Burks), Urbana: Univ. Illinois Press, 1966.
22. M. Gardner, The fantastic combinations of John Conway's new solitaire game 'life,' *Sci. Amer.,* **223** (4): 120–123, 1970.
23. T. Toffoli and N. Margolus, *Cellular Automata Machines—A New Environment for Modeling,* Cambridge, MA: MIT Press, 1987.
24. T. Toffoli and N. Margolus, Invertible cellular automata: A review, *Physica D,* **45**: 1–3, 1990.
25. B. Hasslacher, Discrete fluids, *Los Alamos Sci., Spec. Issue,* **15**: 175–200, 211–217, 1987.
26. U. Frisch et al., Lattice gas hydrodynamics in two and three dimensions, in G. Doolen et al. (eds.), *Lattice-Gas Methods for Partial Differential Equations,* Reading, MA: Addison-Wesley, 1990, pp. 77–135.
27. G. Doolen et al. (eds.), *Lattice-Gas Methods for Partial Differential Equations,* Reading, MA: Addison-Wesley, 1990.
28. D. Rothman, Simple models of complex fluids, in M. Mareschal and B. Holian (eds.), *Microscopic Simulations of Complex Hydrodynamics,* New York: Plenum, 1992.
29. B. Boghosian and T. Washington, Correlations and renormalizations in lattice gases, *Phys. Rev. E,* **52**: 510–554, 1995.
30. J. Yepez, A reversible lattice-gas with long-range interactions coupled to a heat bath, *Fields Inst. Commun.,* **6**: 261–274, 1996.
31. L. Adleman, Molecular computation of solutions to combinatorial problems, *Science,* **266**: 1021–1024, 1994.
32. Insight Institute, *4th Foresight Conf. on Molecular Nanotechnology, Nanotechnology,* **7** (3): 1996.
33. E. Drexler, *Nanosystems: Molecular Machinery, Manufacturing, and Computation,* New York: Wiley, 1992.

33a. T. Knight and G. Sussman, Cellular gate technology, in Ref. 1a.

34. D. Hillis, *The Connection Machine,* Cambridge, MA: MIT Press, 1985.
35. Thinking Machines, *The Connection Machine CM-5 Technical Summary,* Cambridge, MA: Thinking Machines, 1992.
36. C. Leiserson, Fat-trees: Universal networks for hardware-efficient supercomputers, *IEEE Trans. Comput.,* **C-34**: 892–901, 1985.
37. T. Toffoli and N. Margolus, Programmable matter, *Physica D,* **47**: 263–272, 1991.
38. D. Deutsch, *The Fabric of Reality,* New York: Allen Lane, 1997.
39. C. Bennett, The thermodynamics of computation—a review, *Int. J. Theor. Phys.,* **21**: 905–940, 1982.
40. A. Bobeck and H. E. D. Scovil, Magnetic bubbles, *Sci. Amer.,* **224** (6): 78–90, 136, 1971.
41. J. C. Wu, J. P. Hwang, and F. Humphrey, Operation of magnetic bubble logic devices, *IEEE Trans. Magn.,* **20**: 1093–1095, 1988.
42. R. Landauer, Irreversibility and heat generation in the computing process, *IBM J.,* **5**: 183–191, 1961.
43. E. Fredkin and T. Toffoli, Conservative logic, *Int. J. Theor. Phys.,* **21**: 219–253, 1982.
44. C. Bennett, Logical reversibility of computation, *IBM J. Res. Develop.,* **6**: 525–532, 1973.
45. K. Kinoshita, S. Tsutomu, and M. Jun, On magnetic bubble circuits, *IEEE Trans. Comput.,* **C-25**: 247–253, 1976.
46. E. Fredkin and T. Tommaso, Design principles for achieving high-performance submicron digital technologies, proposal to DARPA, MIT Lab. Comput. Sci., MIT, Cambridge, MA, 1978; unpublished but widely circulated and seminal.

46a. M. Frank, T. Knight, and N. Margolus, Reversibility in optimally scalable computer architectures, in Ref. 1a.

47. W. C. Athas et al., Low-power digital systems based on adiabatic-switching principles, *IEEE Trans. VLSI Syst.,* **2**: 398–407, 1994.
48. P. Solomon and D. J. Frank, The case for reversible computation, *Proc. Int. Workshop Low Power Des.,* Napa Valley, CA, 1994, pp. 93–98.
49. S. Younis and T. Knight, Practical implementation of charge recovering asymptotycally zero power CMOS, *Proc. Symp. Integr. Syst.,* 1993, pp. 234–250.
50. P. Shor, Algorithms for quantum computation: Discrete log and factoring, *Proc. 35th Ann. Symp. Found. Comp. Sci.,* IEEE Computer Society, 1994, 116–123.
51. R. Feynman, Simulating physics with computers, *Int. J. Theor. Phys.,* **21**: 467–488, 1982.
52. R. Feynman, Quantum-mechanical computers, *Opt. News,* **11**: 11–20, 1985; reprinted in *Found. Phys.,* **16**: 507–531, 1986.
53. A. Barenco et al., Report on new gate constructions for quantum computation, *Phys. Rev. A,* **52**: 3457, 1995.
54. B. Hayes, The square root of NOT, *Am. Sci.,* **83**: 304–308, 1995.
55. T. Spiller, Quantum information processing: Cryptography, computation, and teleportation, *Proc. IEEE,* **84**: 1719–1746, 1996.
56. C. Williams (ed.), *Quantum Computing and Quantum Communications,* Berlin: Springer-Verlag, 1998.
57. D. Deutsch, Quantum theory, the Church–Turing principle and the universal quantum computer, *Proc. R. Soc. London A,* **400**: 97–117, 1985.
58. D. Hillis, *The connection machine,* MIT Artificial Intelligence Laboratory Memo 646 (1981), substantially reprinted as *The Connection Machine: A Computer Architecture Based on Cellular Automata, Physica D,* **10**: 213–228, 1984.

TOMMASO TOFFOLI
Boston University

NONDESTRUCTIVE MEASUREMENT, MAGNETIC MEANS. See MAGNETIC METHODS OF NONDESTRUCTIVE EVALUATION.

NONDESTRUCTIVE TESTING. See EDDY CURRENT NONDESTRUCTIVE EVALUATION; EDDY CURRENT TESTING.

NONGAUSSIAN PROCESSES. See STATISTICAL SIGNAL PROCESSING, HIGHER ORDER TOOLS.

NONLINEAR CIRCUITS. See TRANSLINEAR CIRCUITS.