Moshe Sipper
Swiss Federal Institute of Technology, Lausanne

The Emergence of Cellular Computing

Von Neumann's complex-processor architecture is the ruling yin of computer technology, but cellular computing promises to be the new yang.

The von Neumann architecture—which is based upon the principle of one complex processor that sequentially performs a single complex task at a given moment—has dominated computing technology for the past 50 years. Recently, however, researchers have begun exploring alternative computational systems based on entirely different principles.

Although emerging from disparate domains, the work behind these systems shares a common computational philosophy, which I call *cellular computing.* This philosophy promises to provide new means for doing computation more efficiently—in terms of speed, cost, power dissipation, information storage, and solution quality. Simultaneously, cellular computing offers the potential of addressing much larger problem instances than previously possible, at least for some application domains.

THREE PRINCIPLES

At its heart, cellular computing consists of three principles: *simplicity*, *vast parallelism*, and *locality*.

Simplicity

The basic processor used as the fundamental unit of cellular computing—the cell—is simple: Although a current, general-purpose processor can perform quite complicated tasks, a cell by itself can do very little. Formally, this notion can be captured by, say, the difference between a universal Turing machine and a finite-state machine. In practice, our field's 50 years of experience in building computing machines gives us a good notion of what "simple" means. For example, an AND gate is simple while a Pentium processor is not.

Vast parallelism

Most parallel computers contain no more than a few dozen processors. In the parallel computing domain, the term *massively parallel* usually describes those few machines that consist of several thousand or, at most, tens of thousands of processors.

Cellular computing involves parallelism on a much larger scale, with the number of cells often measured by the exponential notation 10^x. To distinguish this huge number of processors from that involved in classical parallel computing, I use the term *vast parallelism*. This quantitative difference leads to novel qualitative properties, as Philip Anderson notes in an article on many-body physics whose title, "More Is Different," befits cellular computing as well.[1]

For a more detailed exploration of the differences between cellular and parallel computing, see the "Cellular vs. Parallel Computing" sidebar.

Locality

Cellular computing is also distinguished by its local connectivity pattern between cells. All interactions take place on a purely local basis: A cell can only communicate with a few other cells, most or all of which are physically close by. Further, the connection lines usually carry only a small amount of information. One implication of this principle is that no one cell has a global view of the entire system—there is no central controller.

A different paradigm

Combining these three principles results in the definition *cellular computing* = *simplicity* + *vast parallelism* + *locality*. Because the three principles are highly interrelated, attaining vast parallelism, for example, is facilitated by the cells' simplicity and local connectivity.

Changing any single term in the equation results in a different computational paradigm, as Figure 1's "computing cube" shows. In it, each of the three properties has been placed along one axis. So, for example, foregoing the simplicity property results in the distributed computing paradigm. Cellular computing has been placed further along the parallelism axis to emphasize the "vastness" aspect.

The "Cellular Computing Examples" sidebar shows how cellular computing can be applied to six real-world computing tasks. To illustrate cellular computing's key concepts, I refer to these examples throughout this article.

PROPERTIES OF CELLULAR MODELS

The properties of cellular computing models are highly flexible and can be tailored to specific tasks. Each characteristic that follows presents a choice from several alternatives; the ensemble of these choices results in a specific cellular model.

Cellular versus Parallel Computing

You could argue that the concept of cellular computing is not new at all, but is simply a synonym for parallel computing. Yet the two domains are quite disparate in terms of the models employed and issues studied. Parallel computing has traditionally dealt with a small number of powerful processors, addressing issues such as scheduling, concurrency, message passing, and synchronization. The only conceivable area of intersection between parallel and cellular computing concerns the few so-called "massively parallel machines" built and studied by parallel computing practitioners.[1]

We see that decades of parallel computing research have not produced the expected results—parallel machines are not ubiquitous, and most programmers continue to use sequential programming languages. I believe one reason for this lack of success is the domain's ambitious goal, at least at the outset, of supplanting the serial computing paradigm. However, as parallel computing pioneer Michael J. Flynn recently remarked, "Human reasoning ... is basically a sequential process, although its implementation may be parallel." He noted that "we significantly underestimated the difficulty in achieving the performance speedup expected from parallel processors."[2]

Commenting on the reasons parallel computing has not entirely lived up to its promise, Flynn said: "It is difficult to find large and consistent degrees of parallelism within a single program, because partitioning is too difficult. It is difficult to find parallel algorithms or tools to decompose an existing algorithm into many concurrent tasks." He concluded that "If parallel processors are going to solve this and other problems, the problems' representations should be designed for these machines .... We need to represent problems in a cellular form .... "

Parallel computing teaches cellular computing practitioners that they should not aim for an all-encompassing, general-purpose paradigm that will supplant the sequential one. Rather, we should find those niches where such models can excel. Several clear proofs-of-concept already exist that demonstrate how cellular computing can efficiently solve difficult problems.

Consider a high-speed bullet train that arrives at its destination, only to allow its passengers to disembark through but a single port. This is clearly a waste of the train's parallel exit system, which consists of multiple ports dispersed throughout the train. This metaphor, which I dub the *slow bullet train*, illustrates an important point about parallel systems: their potential (ill-) use in a highly sequential manner. Arthur W. Burks, who completed von Neumann's work on self-reproducing cellular automata,[3] notes: "Thus von Neumann's cellular structure allows for an indefinite amount of parallelism. But in designing his self-reproducing automaton, von Neumann did not make much use of the potential parallelism of his cellular structure. Rather, his self-reproducing automaton works like a serial digital computer .... " This is perhaps the quintessential example of a slow bullet train: embedding a sequential universal Turing machine within the highly parallel cellular-automaton model.

For most cellular computing models, you can prove computation universality by embedding some form of serial universal machine. This proves that *in theory* the model is at least as powerful as any other universal system. However, *in practice* such a construction defeats the purpose of cellular computing by completely degenerating the parallelism aspect. Thus, on the whole, we want to avoid slowing the fast train.

References

1. W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.
2. M.J. Flynn, "Parallel Processors Were the Future ... and May Yet Be," *Computer*, Dec. 1996, pp. 151-152.
3. J. von Neumann, *Theory of Self-Reproducing Automata*, edited and completed by A.W. Burks, University of Illinois Press, Urbana, 1966.

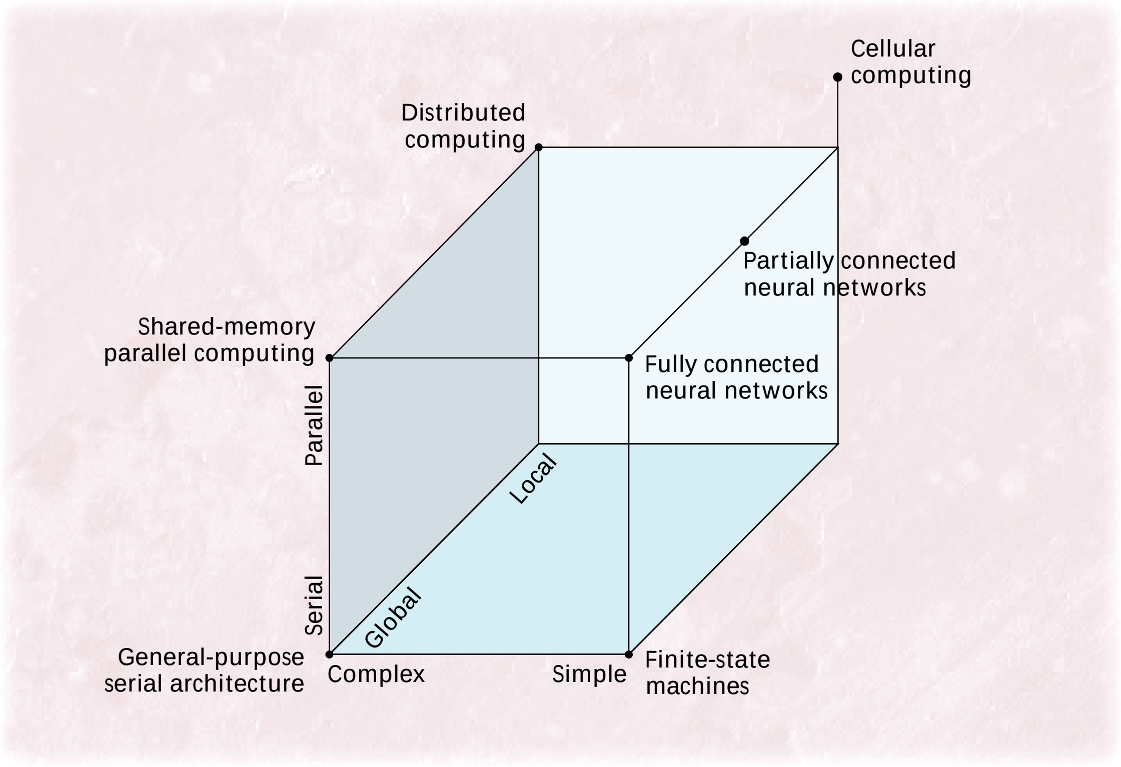
Cellular computing
Distributed computing
Partially connected neural networks
Shared-memory parallel computing
Fully connected neural networks
Parallel
Local
Serial
Global
General-purpose serial architecture
Complex
Simple
Finite-state machines

Figure 1. A graphical representation of the equation simple + vastly parallel + local = cellular computing.

Cell type

Because the cell is the basic unit of computation, we must decide first the exact form it will take:

- *Discrete* cells take on a value from a discrete, finite range of possibilities. These values are often called *states*. The number of possible states may be small or large.
- *Continuous* cells take on an analog value from a given continuous range. The state in this case is analog.

Cell definition

The cell's dynamic behavior over time is defined as a function of its neighbors' values, to which it has access. Any of the following techniques may be used.

- *Exhaustive enumeration* lists the cell's next state for each of the possible neighborhood configurations of states. This technique is usually used for discrete models with a small number of states per cell.
- *Parameterized function* describes the cell's next state as a function of its neighbors. You can distinguish between linear and nonlinear cell functions.
- *Program* computes the cell's next state given the neighboring values.
- *Behavioral rules* specify the cell's behavior in different situations, thus resulting in a specification of its dynamic behavior. These rules may be drawn from, for example, those of molecular biology or quantum physics.

Cell mobility

Cells may either be immobile or mobile. An immobile cell changes only in value. A mobile cell actually moves within a given environment. The self-replicating loops of the satisfiability problem example in the "Cellular Computing Examples" sidebar can be considered mobile at the loop level but immobile at the cellular-automaton level, where only their state values change in time.

Connectivity scheme

Because cells interact with their neighbors, a connectivity scheme must be specified:

- *Regular grid*. A regular grid involves an n-dimensional array (where n = 1,2,3 is used in practice) of a given geometry: rectangular, triangular, hexagonal, and so on. Regularity implies that all cells have the same connectivity pattern. In the cellular neural network example, the grid has a two-dimensional, rectangular geometry, with each cell connected to its eight immediate neighbors, whose state values it can access. When finite grids are involved, you must specify the boundary conditions. Usually, you either assign fixed boundary cells that do not change over time or consider the

grid to be wrapped around itself in a torus-like fashion.
- *Graph.* This is essentially a nonregular grid that can take the form of a directed or undirected graph.

As with the notion of cell simplicity, we resort to arguments from practice for our definition. A fully connected grid, where each cell is connected to each other, also comprises a possible connectivity scheme, but one that falls outside the realm of cellular computing.

Cellular topology or underlying environment

The connectivity scheme applies to the cellular topology itself: The cells are what make up the grid or graph.

Alternatively, you can create an environment for the cells. In this case, the topology need not necessarily be rendered explicitly, as in graph form, but may be given implicitly via a physical or artificial environment that induces spatial contact through random or directed motion. In the DNA-computing example, the physics of molecular motion in a test tube provides an implicit environment, and thus there is no rigid topological structure.

Marco Dorigo and Luca Maria Gambardella[2] provide an example of an explicit underlying environment. Simple mobile cells, called ants, explore in parallel a given graph to solve the traveling salesman problem. In this case, the mobile cells are the ants, and the graph represents not the connectivity of the cells themselves but rather their underlying environment.

Connection lines

As with cells, connection lines are simple, in the sense that the information content transmitted over them is small. A connection might involve, say, merely transmitting the state values of the neighboring cells.

Temporal dynamics

Cellular computing involves systems that change in time at both the cellular and system level. These temporal changes involve two separate choices:

- *Synchrony versus asynchrony.* Synchronous temporal dynamics describe cells that advance in discrete steps, all changing their values simultaneously. Asynchrony means that no such synchronous updating schedule exists. This is not necessarily a black-and-white choice. For example, you may have a melange of both strategies in which all cells within a block are updated simultaneously, while the blocks themselves are updated asynchronously.
- *Discrete versus continuous.* Discrete time describes system behavior in terms of discrete temporal events, be they synchronous or asynchronous. Continuous time means that no such discrete division exists. The cell's behavior may be specified in this latter case by differential operators.

Uniformity

This property refers to the degree of regularity in cellular computing's other properties.

- *Cells.* Are all cells of the same type? Are they identically defined so that they execute the same function? For example, my colleagues and I studied the pseudorandom numbers example with nonuniform cellular automata.[3] In this model,

Cellular Computing Examples

These six examples span a wide range of cellular-computing properties and illustrate the paradigm's key concepts.

Generating pseudorandom numbers with cellular automata

Cellular automata may be the quintessential example of cellular computing, as well as the first to appear on the scene. Conceived in the late 1940s by Stanislaw Ulam and John von Neumann, cellular automata model a dynamic system in which space and time are discrete.[1]

A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states. The cells are updated synchronously in discrete time steps, according to a local, identical interaction rule. The state of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells. This transition is often specified in the form of a *rule table*, which delineates the cell's next state for each possible neighborhood configuration. (The cellular array is n-dimensional, where $n = 1,2,3$ is used in practice.)

Since their inception, cellular automata have been used as a formal model for studying phenomena in several fields, including physics, biology, and computer science. In recent years, interest in using cellular automata as actual computing devices has grown.[2] Pseudorandom number generation is one example of this problem type. Many applications require random numbers, yet finding good random-number generators is difficult. In the past decade, cellular automata have been used to generate random numbers, with their efficacy and viability demonstrated by applying standard tests of randomness.[2]

Cellular adder

In a recent work, Simon Benjamin and Neil Johnson[3] presented a cellular automaton that can perform binary addition: Given two binary numbers encoded as the initial configuration of cellular states, the grid converges in time toward a final configuration that is their sum. This work suggests a possible wireless nanometer-scale realization of the adding cellular automaton that uses coupled quantum dots. As Benjamin and Johnson point out, the device is a nanometer-scale classical computer rather than a different cells may execute different transition functions, as opposed to the original uniform model, in which all cells execute the same function. Recent work indicates that, for a certain class of cellular automata, nonuniformity presents definite computational advantages.[4]

- *Connectivity.* Depending on how a cell connects to its neighbors, the connectivity scheme can be either a regular or nonregular grid.

Determinism versus nondeterminism

In a deterministic model, for any given input the system always goes through the same trajectory of states, ending with the same output. For a nondeterministic system, the same input may result in different trajectories, and possibly different outputs. Nondeterminism may be inherent to the system's functional definition, as with probabilistic and fuzzy systems, or it may result from faults.

OPERATIONAL AND BEHAVIORAL ASPECTS

To understand how a cellular-computing system functions as a whole, you must first grasp seven operational and behavioral aspects of cellular computing.

Programming

Because cellular computing differs from the single-sequential-complex processor model, new programming techniques are required. These techniques can be divided broadly into two categories:

- *Direct programming.* Given a problem to solve, the programmer completely specifies the system at the outset. For example, in cellular automata and cellular neural networks this approach implies not only specifying the cell type, connectivity, and so forth, but also delineating the precise cellular function for each cell. Thus, when the system receives an input that is an instance of the current problem, it computes a correct output. Direct programming does not imply a completely handcrafted system, however, because synthesis tools can be used; rather, it means that the programmer derives a complete specification at the outset. As we shall see, direct programming becomes difficult when global problems are involved.
- *Adaptive methods.* For many problems, the programmer cannot fully specify the system at the outset. In these cases, the programmer only partially specifies the system, which is then subjected to an adaptive process—such as learning, evolution, or self-organization—to produce the desired functionality. For example, nonuniform cellular automata have been evolved using an evolutionary algorithm known as cellular programming.[3] In this case, the programmer determines the basic structure—grid dimensionality, connectivity pattern, and so on—at the outset, then evolves the cellular-transition functions. The connectivity scheme can also be evolved.[3] Both learning methods and evolutionary algorithms have been applied to cellular neural networks to find the cellular-function parameters needed to solve a given problem. Evolutionary algorithms have also been applied in DNA computing, for example, to find good encodings of nucleotide sequences.[5]

true quantum computer. Because it need not maintain wave function coherence, it is far less delicate than a quantum computer.

Solving the contour-extraction problem with cellular neural networks

We can regard a cellular neural network as a cellular automaton in which cellular states are analog rather than discrete, and the time dynamics are either discrete or continuous.[4] The past decade's study has resulted in a lore of both theory and practice, including potentially major application areas such as image processing. For example, in the contour-extraction problem, the network is presented with a grayscale image, and it extracts contours that resemble edges (resulting from large changes in gray-level intensities). This operation, oft-used as a preprocessing stage in pattern recognition, is one example of the many image-processing problems solved by cellular neural networks.

Solving the directed Hamiltonian path problem by DNA computing

Programmers have considered using natural or artificial molecules as basic computational elements—cells—for some time. Leonard Adleman[5] recently gave the decisive proof-of-concept by using molecular-biology tools to solve an instance of the directed Hamiltonian path problem: Given an arbitrary directed graph, you must find whether there exists a path between two given vertices that passes through each vertex exactly once.

Adleman used *oligonucleotides*, short chains of usually up to 20 nucleotides, to encode vertices and edges of a given graph. Next, he placed multiple copies of the oligonucleotides in a real test tube; these oligonucleotides then randomly linked with each other, forming molecules that represent paths through the graph. Adleman then applied molecular-biology procedures to sift through the plethora of candidate DNA-molecule solutions, thus solving the problem—that is, finding out whether a Hamiltonian path exists or not.

The extremely small cell size in this form of cellular computing gives rise to vast parallelism on an entirely new scale. Adleman estimated that such molecular computers could be profoundly faster, more energy efficient, and able to store much more information than current-day supercomputers, The cellular properties described earlier relate to characteristics of the basic system and can be considered as the *first-order dynamics*. Programming, especially via adaptive methods, can be considered a form of *high-order dynamics*, where some of these basic properties change in time.

Local versus global problems

A local problem involves computing a property that can be expressed in purely local terms, as a function of the local cellular neighborhood. Such local operators abound in the domain of low-level image processing, for example, where we define various filters and noise-reducing operations in terms of the cell's nearest neighbors.

I define a global problem in this context as one that involves computation of a nonlocal property.

The difference between global and local problems is not a hard distinction but a matter of degree. Some properties may be only slightly nonlocal, requiring but a small extension beyond the cell's local neighborhood, while other properties may be highly global, in the sense that the entire ensemble of cells must be considered to compute a correct output. Although this distinction between global and local problems is by no means formal, it does provide an idea of the different problem classes that exist. The majority example typifies a global problem in which, to compute a correct output, you must consider the entire grid.

Finding local interaction rules to solve global problems poses a considerable challenge for the system designer. Especially when addressing global problems, it can be difficult to design highly local systems to exhibit a specific behavior. Such problems are prime candidates for adaptive programming methods.

The distinction between local and global can be quite subtle. For example, in the majority example you might consider the naive solution, whereby every cell computes the local majority of its neighboring cells. This does not, however, solve the problem at all since it does not result in computation of the global majority over the entire grid. You can easily solve the majority problem using, for example, a fully connected neural network: A single neuron connected to all input bits, with its threshold properly set, comprises a solution. Thus, in this case, full connectivity is a form of global information.

The issue of local versus global problems relates to *emergent computation*, the appearance of global information-processing capabilities not explicitly represented in the system's elementary components nor in their local interconnections. Currently, emergent computation is ill-defined, with a rigorous, accepted definition still lacking.

Input/output

The I/O question involves two issues. First, as with any problem-solving paradigm, you must specify the set of legal inputs and the desired outputs for the problem at hand. Second, with regard to implementation, you must consider the representation of inputs and outputs in the cellular model, how inputs are presented to the system, and how the output is read out. For example, in the cellular automata solution to the majority example, the input is considered to be the initial configuration of states, and the output is read off the final configuration, in a particular manner.[3] In the cellular adder example, some researchers[6] present

at least with respect to certain problem classes.

You could argue that DNA molecules violate the cell-simplicity principle. However, while such molecules may exhibit complex behavior from the biologist's viewpoint, they can be treated as simple elements from the computational point of view. As Richard Lipton notes, "Our model of how DNA behaves is simple and idealized. It ignores many complex known effects but is an excellent first-order approximation."[6] In DNA computing we typically treat the basic cell, or DNA molecule, as a simple elemental unit on which a few basic operations can be performed in the test tube.[6] This parallels several other instances in computer science where irrelevant low-level details are abstracted away. For example, we usually regard the transistor as a simple switch, considering immaterial the complex atomic and subatomic physical phenomena that underlie its operation.

Solving the satisfiability problem with self-replicating loops

Since the publication of von Neumann's seminal work in the late 1940s,[7] the study of artificial self-replicating structures has produced a plethora of results. Although much of this research has concentrated on theoretical issues, recent studies have raised the possibility of using such self-replicating machines to perform computations.[8]

Hui-Hsien Chou and James Reggia's work,[9] for example, shows that self-replicating structures can be used to solve the NP-complete problem known as satisfiability. Given a Boolean predicate like

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

you must find the assignment of Boolean values to the binary variables x_1, x_2, and x_3 that satisfies the predicate by making it evaluate to True, if such an assignment exists. Chou and Reggia embedded within a two-dimensional, cellular-automaton "universe" simple self-replicating structures in the form of loops. Each loop represents one possible satisfiability solution to the problem; when the loop replicates it gives rise to "daughter" loops, representing different candidate solutions, which go on to self-replicate in their turn. Under a form of artificial selection, replicants representing promising solutions proliferate while those representing failed solutions are lost.

This work can be considered a form of

the cellular automaton's possible implementation as a nanometer-scale device, specifically addressing the I/O issue.

Implementation

While many experiments with cellular-computing systems are done via software simulation, a primary goal is to construct actual machines based on these novel principles. Only then will the full power of the paradigm be realized.

One major implementation advantage involves the low degree of connectivity. As noted by Danny Hillis, "As switching components become smaller and less expensive, we begin to notice that most of our costs are in wires, most of our space is filled with wires, and most of our time is spent transmitting from one end of the wire to the other .... Most of the wires must be short. There is no room for anything else."[7] Another aspect that potentially facilitates implementation is the cells' simplicity.

To date, several novel implementations of "cellware" have been demonstrated, including

- cellular automata implemented as special-purpose digital hardware;
- evolving, nonuniform cellular automata implemented using configurable processors;
- the cellular neural network analog chip;
- molecular computing carried out in actual test tubes; and
- quantum-dot cellular automata, where logic states are encoded not as voltages, as with conventional digital architectures, but rather by the positions of individual electrons.

In cellular computing, implementation issues are often entwined with those concerning the underlying model. That is, you don't design a theoretical model and then consider its implementation, but rather you must address these two issues simultaneously. Choosing the basic properties of the model has immediate consequences on implementation, and vice versa—often you start with implementation constraints that have implications where the underlying model is concerned. Indeed, this interplay between model properties and system aspects is characteristic of cellular computing in general.

Scalability

Cellular computing offers a paradigm potentially more scalable than classical models, thanks to local connectivity, the absence of a central processor that must communicate with every single cell, and the simplicity of the cells themselves. Thus, prima facie, adding cells should not pose a major problem. In reality, however, this issue is not trivial at either the model or implementation level. As I note elsewhere,[3] *simple* scaling, which involves a straightforward augmentation in resources such as cells and connections, does not necessarily bring about *task* scaling, defined as the maintenance of at least the same performance level for the problem at hand.

Robustness

A system's robustness derives from its ability to function in the face of faults. When cells function incorrectly, communication links fail, or some other mishap occurs, we want the system to continue functioning, or at least exhibit graceful degradation.

The attributes of cellular systems give rise, in many

DNA computing in a cellular automaton, using self-replicating loops in a vastly parallel fashion. A molecular implementation of this approach might be had by using recently created synthetic self-replicators. Lipton[6] presented a DNA-computing solution to the satisfiability problem that's similar to Adleman's method, noting that "biological computations could potentially have vastly more parallelism than conventional ones."

Solving the majority problem with cellular automata

In this problem, a binary-state cellular automaton should classify an arbitrary initial configuration of states (the input) by whether there is a majority of 0s or 1s. Since majority is a global configuration property (the 1s can be distributed throughout the entire grid) whereas the cellular automaton is highly local (that is, no single cell can directly compute the answer by itself), the problem is not trivial. Several researchers have studied it over the past decade, applying evolutionary computation techniques to evolve solutions.[2]

References

1. J. von Neumann, *Theory of Self-Reproducing Automata*, edited and completed by A.W. Burks, University of Illinois Press, Urbana, 1966.
2. M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, Springer-Verlag, Heidelberg, 1997.
3. S.C. Benjamin and N.F. Johnson, "A Possible Nanometer-Scale Computing Device Based on an Adding Cellular Automaton," *Applied Physics Letters*, 1997, pp. 2,321-2,323.
4. L.O. Chua and T. Roska, "The CNN Paradigm," *IEEE Trans. Circuits and Systems*, Mar. 1993, pp. 147-156.
5. L.M. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, 1994, pp. 1,021-1,024.
6. R.J. Lipton, "DNA Solution of Hard Computational Problems," *Science*, 1995, pp. 542-545.
7. M. Sipper et al., eds., special issue on Artificial Self-Replication, *Artificial Life*, 1998.
8. M. Sipper et al., "A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems," *IEEE Trans. Evolutionary Computation*, 1997, pp. 83-97.
9. H.-H. Chou and J.A. Reggia, "Problem Solving During Artificial Selection of Self-Replicating Loops," *Physica D*, 1998, pp. 293-312.

cases, to increased resilience. Local connectivity allows for easier fault containment by confining the fault to one region, preventing its easy spread throughout the system, while the vast number of extant cells enables a large part of the system to remain operational.

Hierarchy

Hierarchical decomposition is ubiquitous in both natural and artificial systems.

In computer science there is the *programming* hierarchy: high-level language, intermediate code, assembly language, machine language, and the transistor level. Such a hierarchy facilitates the end-user's task enormously.

In nature we find *organizational* hierarchies, such as the molecule-cell-organ infrastructure, as well as *processing* hierarchies, such as the human visual system, which begins with low-level image processing in the retina and ends with high-level operations—such as face recognition—performed in the cortical regions.

Such hierarchical forms also have their place in cellular computing, being either fixed at the outset or emerging through adaptive programming. As an example, consider self-replicating loops, where we can distinguish between two levels: the lower, cellular-automaton level and the higher, self-replicating loops level. These levels can be considered separately, and you can even imagine a different low-level implementation using synthetic molecules. Thus, we can consider the two levels a programming hierarchy. Furthermore, the levels in this example exhibit different behaviors: The cellular-automaton level consists of immobile cells that operate in synchronous mode, while the loop level consists of mobile cells—the loops—that can be considered to operate asynchronously.

Cellular computing has attracted increasing research interest recently. Work in this field has produced exciting results that hold prospects for a bright future. Yet several questions must be answered before cellular computing can become a mainstream paradigm. What classes of computational tasks are most suited to it? Can these be formally defined, or informal guidelines established? How do we match the specific properties and behaviors of a given model to a suitable class of problems?

Once we derive a baseline from the answers to these questions, we can extend cellular computing's programming methodologies and possibly introduce novel ones. Evolution, learning, and self-organization have been shown viable. More recently, the use of *ontogeny*—the developmental process of a multicellular organism—has also been explored.[8]

What specific application areas invite a cellular-computing approach? Research has raised several possibilities:

- *Image processing.* Applying cellular computers to perform image-processing tasks arises as a natural consequence of their architecture. For example, in a two-dimensional grid, a cell (or group of cells) can correspond to an image pixel, with the machine's dynamics designed to perform a desired image-processing task. Research has shown that cellular image processors can attain high performance and exhibit fast operation times for several problems.
- *Fast solutions to NP-complete problems.* Even if only a few such problems can be dealt with, doing so may still prove highly worthwhile. NP-completeness implies that a large number of hard problems can be efficiently solved, given an efficient solution to any one of them. The list of NP-complete problems includes hundreds of cases from several domains, such as graph theory, network design, logic, program optimization, and scheduling, to mention but a few.
- *Generating long sequences of high-quality random numbers.* This capability is of prime import in domains such as computational physics and computational chemistry. Cellular computers may prove a good solution to this problem.
- *Nanoscale calculating machines.* Cellular computing's ability to perform arithmetic operations raises the possibility of implementing rapid calculating machines on an incredibly small scale. These devices could exceed current models' speed and memory capacity by many orders of magnitude.
- *Novel implementation platforms.* Such platforms include reconfigurable digital and analog processors, molecular devices, and nanomachines.

As we work to shape this new paradigm, it is encouraging to consider that nature is cellular computing's ultimate proof-of-concept.

For cellular computing to become viable, it must prove its scalability. This issue is still in doubt, as Lipton noted when he observed that Adleman "solved the HPP [Hamiltonian Path Problem] with brute force: He designed a biological system that 'tries' all possible tours of the given cities."[9] Yet even if you use 10^{23} parallel computers, Lipton warns, you "cannot try all tours for a problem with 100 cities. The brute force algorithm is simply too inefficient."[10]

Scalability also relates to the hierarchy question. For example, when either artificial or natural systems are scaled up by orders of magnitude, a hierarchical structure is usually imposed, be it programming, organizational, processing, or some other form.

Lipton[10] addresses the issues of implementation and robustness as well, citing the errors generated by less-than-perfect operation as a possible barrier to building DNA computers. This imperfection motivated Russell Deaton and colleagues[5] to apply evolutionary techniques that search for better DNA encodings, thus reducing the errors during DNA computation. In my own work,[3] I've studied the effects of random faults on the behavior of some evolved cellular automata, showing that they exhibit graceful degradation in performance and can tolerate a certain level of faults.

Many natural systems exhibit striking problem-solving capacities, while adhering to the principles of cellular computing. There may be a two-way street: We can seek inspiration in natural processes, creating novel cellular methodologies; at the same time, research into cellular computing, though mainly an engineering activity, may increase our understanding of computation in nature. Indeed, as we work to shape this new paradigm, it is encouraging to consider that nature is cellular computing's ultimate proof-of-concept. ❖

Acknowledgments

I thank Mathieu Capcarrère, Daniel Mange, Eduardo Sanchez, Marco Tomassini, and the anonymous reviewers for their many helpful remarks and suggestions. This work was supported in part by grant 2000-049349.96 from the Swiss National Science Foundation.

References

1. P.W. Anderson, "More Is Different," *Science*, 1972, pp. 393-396.
2. M. Dorigo and L.M. Gambardella, "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem," *IEEE Trans. Evolutionary Computation*, 1997, pp. 53-66.
3. M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, Springer-Verlag, Heidelberg, 1997.
4. M. Sipper, "Computing with Cellular Automata: Three Cases for Nonuniformity," *Physical Review E*, 1998, pp. 3,589-3,592.
5. R. Deaton et al., "A DNA Based Implementation of an Evolutionary Search for Good Encodings for DNA Computation," *Proc. 1997 IEEE Int'l Conf. Evolutionary Computation* (*ICEC 97*), IEEE Press, Piscataway, N.J., 1997, pp. 267-271.
6. S.C. Benjamin and N.F. Johnson, "A Possible Nanometer-Scale Computing Device Based on an Adding Cellular Automaton," *Applied Physics Letters*, 1997, pp. 2,321-2,323.
7. W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.
8. M. Sipper et al., "A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems," *IEEE Trans. Evolutionary Computation*, 1997, pp. 83-97.
9. L.M. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, 1994, pp. 1,021-1,024.
10. R.J. Lipton, "DNA Solution of Hard Computational Problems," *Science*, 1995, pp. 542-545.

Moshe Sipper is a senior researcher in the Logic Systems Laboratory at the Swiss Federal Institute of Technology, Lausanne. His chief interests involve the application of biological principles to artificial systems, including evolutionary computation, cellular computing, bio-inspired systems, evolving hardware, complex adaptive systems, artificial life, and neural networks. Sipper has published close to 70 papers in these areas and the book Evolution of Parallel Cellular Machines: The Cellular Programming Approach *(Springer-Verlag, 1997). Sipper received a BA in computer science from the Technion-Israel Institute of Technology and an MSc and a PhD in computer science from Tel Aviv University.*

Contact Sipper at Moshe.Sipper@epfl.ch.