Physica D 42 (1990) 1–11
North-Holland

EMERGENT COMPUTATION: SELF-ORGANIZING, COLLECTIVE, AND COOPERATIVE PHENOMENA IN NATURAL AND ARTIFICIAL COMPUTING NETWORKS

INTRODUCTION TO THE PROCEEDINGS OF THE NINTH ANNUAL CNLS CONFERENCE

Stephanie FORREST
Center for Nonlinear Studies and Computing Division, MS-B258, Los Alamos National Laboratory, Los Alamos, NM 87545, USA

Parallel computing has typically emphasized systems that can be explicitly decomposed into independent subunits with minimal interactions. For example, most parallel processing systems achieve speedups by identifying code and data segments that can be executed simultaneously. Under this approach, interactions among the various segments are managed directly, through synchronization, and communication among components is viewed as an inherent cost of computation. As a result, most extant parallel systems require substantial amounts of overhead to manage and coordinate the activities of the various processors, and they obtain speedups that are considerably less than a linear function of the number of processors.

An alternative approach exploits the interactions among simultaneous computations to improve efficiency, increase flexibility, or provide a more natural representation. Researchers in several fields have begun to explore computational models in which the behavior of the entire system is in some sense more than the sum of its parts. These include connectionist models [46], classifier systems [22], cellular automata [5, 7, 56], biological models [11], artificial-life models [33], and the study of cooperation in social systems with no central authority [2]. In these systems interesting global behavior *emerges* from many local interactions. When the emergent behavior is also a computation, we refer to the system as an *emergent computation*.

The distinction between standard and emergent computations is analogous to the difference between linear and nonlinear systems. Emergent computations arise from nonlinear systems while standard computing practices focus on linear behaviors (see section 2.2). The idea that interactions among simple deterministic elements can produce interesting and complex global behaviors is well accepted in the physical sciences. However, the field of computing is oriented towards building systems that accomplish specific tasks, and emergent properties of complex systems are inherently difficult to predict. Thus, it is not immediately obvious how architectures (either hardware or software) that have many interactions with often unpredictable effects can be used effectively, and it is for this reason that I have chosen to use the term "emergent computation" instead of referring more broadly to nonlinear properties of computational systems. The premise of emergent computation is that interesting and useful computational systems can be constructed by exploiting interactions among primitive components, and further, that for some kinds of problems (e.g. modeling intelligent behavior) it may be the only feasible method.

To date, there has been no unified attempt to specify carefully what constitutes an emergent computation or to determine what properties are required of the supporting architectures that generate them. This volume contains the Proceedings of a recent Conference held at Los Alamos National Laboratory devoted to these questions. Emergent computation is potentially relevant to several areas, including adaptive systems, parallel processing, and cognitive and biological modeling, and the Proceedings reflects this diversity, with contributions from physicists, computer scientists, biologists, psychologists, and philosophers. The Proceedings are thus intended for an interdisciplinary audience. Each paper addresses this by providing introductory explanations beyond that required for a field-specific publication. In this introduction I hope to stimulate discussion of emergent computation beyond the scope of the actual conference. First, a definition is proposed and several detailed examples are presented. Then several common themes are highlighted, and the contents is briefly reviewed.

1. What is emergent computation?

It is increasingly common to describe physical phenomena in terms of their information processing properties [57, 58]. However, we wish to distinguish emergent computation from the general emergent properties of complex phenomena. We do this by requiring that both the explicit and the emergent levels of a system be computations. For example, a Rayleigh–Bénard convecting flow, in which the dynamics of the fluid particles follow a chaotic path, would not necessarily be considered a form of emergent computation.

The requirements for emergent computation are quite similar to those proposed by Hofstadter in his paper on subcognition [19]. He stresses that information which is absent at lower levels can exist at the level of collective activities. This is the essence of the following constituents of emergent computation:

(i) A collection of agents, each following explicit instructions;

(ii) Interactions among the agents (according to the instructions), which form implicit global patterns at the macroscopic level i.e. epiphenomena;

(iii) A natural interpretation of the epiphenomena as computations.

The term "explicit instructions" refers to a primitive level of computation, also called "micro-structure", "low-level instructions", "local programs", "concrete structure", and "component subsystems". In a typical case, such as cellular automaton, each cell acts as an agent executing the instructions in its state-transition table. However, in some cases the coding for an instruction may not be distinguished from the agent that executes it. The important point is that the explicit instructions are at a different (and lower) level than the phenomena of interest. The level of an instruction is determined by the entity that processes it. For example, if the low-level instructions were machine code, they would be executed by hardware, while higher-level instructions would be interpreted by "virtual machines" simulated by the lower-level machine code instructions. The higher-level instructions would be implicit, although not necessarily the product of interactions (see section 2.2 on superposition).

There is a tension between low-level explicit computations and the patterns of their interaction, and the interaction among the levels is important. Global patterns may influence the behavior of the lower-level local instructions, that is, there may be feedback between the levels. Patterns that are interpretable as computations process information, which distinguishes emergent computation from the interesting global properties of many complex systems such as the Rayleigh–Bénard experiment mentioned earlier.

Central to the definition is the question of to what extent the patterns are "in the eye of the beholder", or interpreted, and to what extent they are inherent in the phenomena itself. This issue arises because the phenomena of interest are implicit rather than explicit. Note that to a lesser extent the interpretation problem also exists in standard computation. The difference is that in emergent computations there is no one address (or set of addresses) where one can read out an answer. Thus, the time for interpretation is likely to be much lower for standard computations than emergent ones. Currently, many emergent computations are interpreted by the perceptual system of the person running the experiment. Thus, when conducting a cellular automaton experiment, researchers typically rely on graphics-based simulations to reveal the phenomena of interest. While quantitative measures can be developed in some cases to interpret the results, scientific visualization techniques are an integral part of most current emergent computations.

According to the Church–Turing thesis, a Turing machine can both implement any definable computation and simulate any set of explicit instructions we might choose as the basis of an emergent computation [23]. Thus, the concept of emergent computation cannot contribute magical computational properties. Rather, we are advocating a way of thinking about the design of computational systems that could potentially lead to radically different architectures which are more robust and efficient than current designs[#1].

A related question is whether or not emergent computations can be implemented in more traditional ways (i.e. can the emergent patterns be encoded as a set of explicit instructions instead of indirectly as implicit patterns?). While in some cases it may be possible to encode the emergent patterns directly in some language or machine, there are several advantages to an emergent-computation approach, including efficiency, flexibility, representation, and grounding. First, implementing computations indirectly as emergent patterns may provide implementation efficiencies because of the need for less control over the different components (e.g. processes). As mentioned earlier, a high proportion of computing time is devoted to managing interactions among processes. Other kinds of efficiencies may also be realized, including efficiencies of cost through the use of multiple cheap components, efficient uses of programmer time, and raw computational speed through the use of massive parallelism. Second, flexibility is important for systems that must interact with complex and dynamic environments, e.g. intelligent systems. For these systems, it is impossible to get enough flexibility from explicit instructions; for realistic environments, it is just not possible to program in all contingencies ahead of time. Therefore, the flexibility must appear at the emergent level. The interaction between the instructions and the environment (or between emergent properties of the instructions and the environment) is important, and there are global patterns (symbols, etc.) associated with this instruction–environment interaction. Third, the advantage in representation arises in systems for which it is difficult to articulate a formal description of the emergent level. Several authors have argued for the impossibility of such an undertaking for systems of sufficient complexity such as weather patterns and living systems [19, 33, 35, 48]. In these circumstances, emergent systems may provide the most natural model. Finally, the grounding issue arises if the emergent patterns are intended as real phenomena or models of real phenomena (as in cognitive modeling). In this circumstance, the intended interpretation of a purely formal model (e.g. symbolic models of artificial intelligence) becomes problematic since the model is not connected to (grounded in) the domain of interest (by e.g. a sensory interface). Emergent-computation models can address this problem by using low-level explicit instructions that are directly connected to the domain. Harnad's paper discusses the grounding problem in detail [17].

At the architectural level, there are two criteria that capture the spirit of emergent computation: efficacy and efficiency. The criterion of computational efficacy is met by systems in which each

[#1] Even if in principle emergent computations can be simulated by a Turing machine, interpreting the resulting patterns as computations is likely to be so difficult as to be infeasible.

computational unit has limited processing power (e.g. a finite state machine) and in which the collective system is computationally more powerful (e.g. a Turing machine). The criterion of computational efficiency can be met by parallel models that are capable of linear or better than linear speedups relative to the number of processors used to solve the problem. (Other criteria can be imagined that are less strict. For example, it may be reasonable to construct systems in which not all of the processors are used all of the time. In these cases, the dimensions of time and number of processors might be combined to obtain a reasonable definition somewhat different from that mentioned above.) Previous work on computational systems with interesting collective/emergent properties has generally focused on some variant of the first of these criteria and ignored the second.

2. Example problems

In this section, three concrete examples are presented to illustrate what sorts of problems emergent computation can address and provide a framework for interpreting the definition. The parallel processing example shows how emergent computation can lead to efficiency improvements. The section on programming languages establishes the connection between emergent computation and nonlinear systems. Finally, two search techniques are compared to show how the emergent-computation approach to a problem differs from other more conventional approaches.

2.1. Parallel processing

Consider the problem of designing a large complex computational system to perform reliably and efficiently. By "large complex" we mean that there are many components and many interactions among the components. The computational system could be a complicated algorithm that we would like to run in parallel, it could be distributed over many machines with possibly heterogeneous operating systems, or it could simply be a large, evolving software package being modified simultaneously by several different programmers. In the following, we will focus on the parallelization example, but similar arguments can be made for the distributed systems and software engineering aspects of the problem.

The conventional approach to such a problem looks for code and/or data segments that can be executed independently, and hence simultaneously. With this view, it is important to minimize the interactions among the various components. Synchronization strategies are defined to manage communication among the independent components. The controlling program *knows* about all possible interactions and manages them directly. Thus, interactions are viewed as costs to be minimized. As mentioned earlier, most extant parallel systems consume substantial amounts of overhead managing and coordinating the activities of their processors. This is because the flow of data and control rarely match exactly the interconnection topology of the parallel machine. Potential speedups are also limited by inherently sequential components within a computation, a phenomenon quantified by Amdahl's law [1]. For these reasons, the speedups that are achieved by most parallel systems are considerably less than a linear function of the number of processors.

Emergent computation suggests a view of parallelism in which the interactions among components lead to problem solutions with potentially better than linear performance. For example, a system that performs explicit search at the concrete architecture level, but is implicitly searching a much larger space (see section 2.3) meets this criterion.

The claim that superlinear speedups are in principle possible is controversial. The standard counterargument is as follows: if a process P runs in t time on n processors, then there exists a sequential machine that can simulate P in at most $(k_1 nt) + k_2$ time steps, where k_1 and k_2 are constants, so the speedup is only by a factor of n. This counterargument ignores the time required to interpret the results. If the result of the computation is a global pattern (e.g. a pattern of states in a cellular automaton distributed across several time steps), then the procedure for recognizing that same pattern on a sequential machine might require as much computation as the original computation.

Even allowing for the theoretical possibility of superlinear speedups, one might question whether or not it is feasible to actually construct such a system. The following simple example illustrates how interactions among components can provably help the efficiency of a computation. In formal models of parallel computation, there are various assumptions about what happens when two independent processors try to write to the same location in global memory simultaneously. Some of these assumptions forbid any interaction between the two simultaneous writes: for example, one processor is allowed to dominate and write successfully while the other processor is forced to wait (an "Exclusive Write"). Others exploit the interaction: for example, one version of the "Concurrent Write" model prevents both processors from writing but records the collision as a "?" in the memory cell, destroying the previous contents. The Concurrent Write model turns out to be provably stronger than the Exclusive Write in the sense that certain parallel algorithms can be implemented more efficiently with Concurrent Write than they can with Exclusive Write (for example, computing certain kinds of disjunction [24]). The trick is that if the interaction is recorded as a "?" then both processors that tried to write can inspect that memory cell and determine that there was a collision. This information can be exploited in certain circumstances to produce more efficient algorithms. Thus, in the Concurrent Write model, write collisions (interactions) are shown to be a useful form of computation, leading to performance improvements, even if the collisions themselves do not result in transmitted values reaching their destination. This small example meets the criterion of "computing by interaction", although the interactions are recorded explicitly rather than implicitly, as we would expect in a truly emergent computation.

Generally, we expect the emergent-computation approach to parallelism to have the following features: (1) no central authority to control the overall flow of computation, (2) autonomous agents that can communicate with some subset of the other agents directly, (3) global cooperation (see section 3) that emerges as the result of many local interactions, (4) learning and adaptation replacing direct programmed control, and (5) the dynamic behavior of the system taking precedence over static data structures.

2.2. *Programming languages and the superposition principle*

Emergent computation arises from interaction among separate components. There are several ways in which the standard approach to programming-language design minimizes the potential for emergent computation. This example explores the connection between emergent computation and nonlinear systems.

The notation, or syntax, used to express computer programs is for the most part *context free*. Roughly, this means that legal programs are required to be written in such a way that the legality (whether or not the program is syntactically correct) of any one part of the program can be determined independently of the other parts. While this is a very powerful property (among other things, it makes it possible to build efficient compilers), emergent computations are almost certainly not context free since they arise from interactions among components. However, the low-level instructions that generate emergent computations may well be context free.

The semantics of programming languages can be described by any of several different standard mathematical models [52]. These models describe how the syntax should be interpreted, that is, what a program *means* (more specifically, what function it computes). The meaning of a program helps determine the set of low-level machine instructions that are executed when a program *runs*. The standard approaches to programming-language semantics discourage emergent computation. For example, in the denotational semantics approach [52], the meaning of a program is determined by composing the meanings of its constituents. The meaning of an arithmetic expression $A + B$ might be written as follows: (read $[\![\mathit{expression}]\!]$ as "the meaning of expression")[#2]:

$$[\![A \oplus B]\!] = [\![A]\!] + [\![B]\!].$$

Thus, the meaning of A is isolated from B, and can be computed independently of it. Similar expressions can be written for all the common programming constructs, including assignment statements, conditional statements, and loops. By contrast, we expect that in an emergent computation there would be interactions between components that would not interact in standard computation.

This compositional approach to programming-language semantics is analogous to the superposition principle in physics, which states that for homogeneous linear differential equations, the sum of any two solutions is itself a solution. Systems that obey the superposition principle are linear and thus incapable of generating complicated behaviors associated with nonlinear systems such as chaos, solitons, and self-organization. Similarly in the domain of programming languages, the ability to define the meaning of context-free programs in terms of their constituent parts indicates that there are few if any interactions between the meaning of one part and the meaning of another. In these sorts of languages and models, the goal is to minimize side effects that could lead to inadvertent interactions (e.g. changing the value of a global variable) – once again, emergent computation is primarily computation by side effect. The analogy between the superposition principle in physics and the compositional approach of denotational semantics suggests that something like the distinction between linear and nonlinear models in physics exists in computational systems. Note, however, that it is possible to write programs in context-free languages that have nonlinear behaviors when executed, e.g. a simple logistic map, just as it may be possible to write the low-level instructions for an emergent-computation system in a context-free language.

While nonlinear computational systems are more difficult to engineer than linear ones, they are capable of much richer behavior. The role of enzymes in catalysis provides a nice example of how nonlinear effects can arise from simple recombinations of compounds [10]. More generally, consider the problem of recombination in adaptive systems. If one can detect combinations that yield effects not anticipated by superposition, then those combinations can be exploited in various ways that are not available in a model based on principles of superposition.

A final example of how the principle of superposition pervades standard programming languages is provided by the Church–Rosser theorem [9]. The λ calculus defines a formal representation for functions and is closely related to the Lisp programming language. In the λ calculus, various substitution and conversion rules are defined for reducing λ expressions to normal form. The Church–Rosser theorem (technically, one of its corollaries) shows that no λ expression can be converted to two different normal forms (for example, by applying reductions in different order). This is another example of how computer science gets a lot of leverage out of systems that have something like the principle of superposition. Since nonlinear systems often have the property that operations applied in different orders have different effects, emergent computations will not in general have nice simplification rules like the Church–Rosser theorem.

2.3. Search

The problem of searching a large space of possibilities for an acceptable solution, a particular

[#2]Two different plus symbols are used to distinguish between the symbol plus ($\oplus$) and the operation that implements it ($+$).

datum, or an optimal value is one of the most basic operations performed by a computer. Intelligent systems are often described in terms of their capabilities for "intelligent" search – that is the ability to search an intractably large space for an acceptable solution, using knowledge-based heuristics, previous experience, etc. The various techniques of intelligent search provide a sharp contrast between emergent computation and traditional approaches to computation. A classical approach to the problem of search is that of an early artificial intelligence program, the general problem solver (GPS) [39], while the emergent-computation approach is illustrated by the genetic algorithm [20].

GPS uses means–ends analysis to search a state space to find some predetermined goal state. GPS works by defining subgoals part way between the start state and the goal state, and then solving each of the subgoals independently (and recursively). Under this approach, the domain of problem solving is viewed as "nearly decomposable" [50], meaning that for the most part each subgoal can be solved without knowledge of the other subgoals in the system. The overall approach taken by GPS is still prevalent in artificial intelligence, the recent work on SOAR [31] being a good example.

In contrast, genetic algorithms [14, 20] show how emergent computation can be used to search large spaces. There are two levels of the algorithm, explicit and implicit. At the mechanistic or explicit level, a genetic algorithm consists of:

(i) A population of randomly chosen bit strings: $P \subseteq \{0,1\}^l$, representing an initial set of guesses, where l is a fixed positive integer denoting the length in bits of a guess;

(ii) A fitness function: F: *guesses* $\rightarrow \mathbb{R}$, where $\mathbb{R}$ denotes the real numbers;

(iii) A scheme for differentially reproducing the population based on fitness, such that more copies are made of more fit individuals and fewer or no copies of less fit ones;

(iv) A set of "genetic" operators (e.g. mutation, crossover, and inversion) that modify individuals to produce new guesses;

(v) Iteration for many generations of the cycle: evaluation of fitness, differential reproduction, and application of operators. Over time, the population will become more like the successful individuals of previous generations and less like the unsuccessful ones.

At the virtual, or implicit, level, we can interpret the genetic algorithm as searching a higher-order space of patterns, the space of hyperplanes in $\{0,1\}^l$. When one individual is evaluated by the fitness function, many different hyperplanes are being sampled simultaneously – the so-called implicit parallelism of the genetic algorithm. For example, evaluating the string 000 provides information about the following hyperplanes[#3]:

000, 00#, 0#0, #00, 0##, #0#, ##0, ###.

Populations undergoing reproduction and cross-over (with some other special conditions) are guaranteed exponentially increasing samples of the observed best schemata (a property described in refs. [14, 20]). Thus, performance improvements provably arise from the collective properties of the individuals in the population over time. The population serves as a distributed database that implicitly contains recoverable information about the multitudes of hyperplanes (because each individual serves in the sample set of many hyperplanes). Put another way, the population reflects the ongoing statistics of the search over time.

Several aspects of emergent computation are illustrated by this example. The algorithm is very flexible, allowing it to track changes in the environment. Since the statistical record of the search is distributed across the population of individuals, interpretation is an issue if there is a need to recover the statistics explicitly. Normally it is sufficient to look at a few typical individuals or to

[#3]The # symbol means "don't care". Thus, #00 denotes the pattern, or *schema*, which requires that the second 2 bits be set to 0 and will accept a 0 or a 1 in the first bit position. The space of possible schemata is the space of hyperplanes in $\{0,1\}^l$. (See ref. [14] for an introduction to both the mechanism and theory of the genetic algorithm.)

treat the best individual seen as the "answer" to the problem. The potential efficiency of emergent computation is also demonstrated through the use of implicit parallelism. There is a price, however. While the algorithm is highly efficient, it achieves its efficiency through sampling. This means that there is some loss of accuracy (see Greening's paper in these Proceedings [15] for a careful treatment of this issue).

3. Themes of emergent computation

Three important and overlapping themes of systems that exhibit emergent computation are self-organization, collective phenomena, and cooperative behavior. Here, we use the term self-organization to mean the spontaneous emergence of order from an initially random system, but see ref. [40] for a detailed formulation of self-organization. Collective phenomena are those in which there are many agents, many interactions among the agents, and an emphasis on global patterns. A third component of emergent computation is the notion of cooperative behavior, i.e. that the whole is somehow more than the sum of the parts. In this section, these three themes are illustrated in the context of several examples.

One of the most compelling examples comes from nature in the form of ant colonies. The actions of any individual ant are quite limited and apparently random, but the collective organization and behavior of the colony is highly sophisticated, including such activities as mass communication and nest building [53, 54]. In the absence of any centralized control, the collective entity (the colony) can "decide" (the decision itself is emergent) when, where, and how to build a nest – self-organizing, collective, and cooperative behavior in the extreme. Clearly, many of the activities in an ant colony involve information-processing, such as laying trails from the nest to potential food sites, communicating the quality and quantity of food at a particular site, etc. By making an analogy between the cells in a cellular automaton and individual ants, Langton has described computational models that emulate some of the important information-processing aspects of ant colonies [32].

Kauffman's article in these Proceedings [28] explores self-organizing behavior in simple randomly connected networks of Boolean function. These networks spontaneously organize themselves into regular structures of "frozen components" that are impervious to fluctuating states in the rest of the network. The tendency of a network to exhibit this and other self-organizing behaviors is related to various structural properties of the network and more generally to the problem of adaptation.

Not all examples of emergent computation are beneficial. The Internet (a nationwide network for exchanging electronic mail) was designed so that messages would be routed somewhat randomly (there are usually many different routes that a message may take between two Internet hosts). The intent is for message traffic to be evenly distributed across the various hosts. However, in some circumstances the messages have been found to self-organize into a higher-level structure, called a token-passing ring, so that all of the messages collect at one node, and then are passed along to the next node in the ring [26]. In this case, the self-organization is highly detrimental to the overall performance of the network. The behavior raises the question of what, if any, low-level protocols could reliably prevent harmful self-organizing behavior in a system like the Internet.

In a computational setting, there are at least two quite different types of cooperation: (1) program correctness, and (2) resource allocation. In this context, program correctness means that a collection of independent instructions evolves (more accurately, coevolves) over time in such a way that their interactions result in the desired global behavior. That is, the adaption takes place at the instruction level, but the behavior of interest is at the collective level. If the collective instructions (a program) learn the correct behavior, we say that they are cooperating. Holland's classifier systems (see papers in these Proceedings) are a good example of this sense of cooperation. The second meaning of cooperation occurs when some shared resource on a local area network (e.g., CPU time, printers, network access, etc.) is allocated efficiently among a set of distributed processes. The Huberman and Kephart et al. papers in this volume [24, 30] discuss how robust resource-allocation strategies can emerge in distributed systems.

4. Review of contents

This introduction has described one view of emergent computation. The conference produced several themes and topics of its own. In particular, the themes of design (how to construct such systems), learning and the importance of preexisting structure, the role of parallelism, and the tension between cooperative and competitive models of interaction are central to many of the papers in the Proceedings. Emergent-computation systems can be constructed either by adapting each individual component independently or by tinkering with all of the components as a group. Wilson's paper [55] addresses this issue of local- versus system-level design. Learning is clearly central to emergent computation, since it provides the most natural way to control such a system. Several papers in the Proceedings (Baird, Banzhaf and Haken, Hansen, Omohundro, Schaffer et al. [3, 4, 16, 41, 47]) focus on specific learning issues, and many others use learning as an integral part of their system. The role of parallelism in emergent computation is often assumed. However, Machlin and Stout's paper [36] challenges that assumption, and Greening's paper [15] explores the consequences of using parallelism efficiently.

The papers have been grouped roughly into the following subject areas: (1) artificial networks, (2) learning and adaptation, and (3) biological networks. Thus, all of the papers on biological networks are grouped together, although they emphasize different aspects of problems of emergent computation.

There is a wide range of papers concerned with emergent behavior and computing. Langton's paper [34] illustrates the importance of phase transitions to emergent computation. Huberman [24], Kephart et al. [30], and Maxion [37] discuss emergent behaviors in computing networks. Machlin and Stout's paper [36] illustrates how very simple Turing machines can exhibit interesting and complex behavior. Palmore and Herring's paper [42] provides an example of the connection between emergent computation and real computing procedures (computer arithmetic). Rasmussen's paper [44] uses a simple model of computer memory to show how cooperative "life-like" structures can emerge under various conditions. Finally, Kauffman's paper [28] explores the self-organizing properties of simple Boolean networks.

The adaptive systems aspect of emergent computation is a dominant theme in the Proceedings, and Farmer's paper [10] relates various models of learning through the common thread of adaptive dynamics. Papers on classifier systems and genetic algorithms range from proposals for new mechanisms (Holland [21]) to methods for analyzing classifier system behavior (Compiani et al. [8] and Forrest and Miller [12]), to bridges between genetic algorithms and neural networks (Schaffer et al. [47] and Wilson [55]). Two papers (Hillis [18], Ikegami and Kaneko [25]) explore how interactions between hosts and parasites can improve the global behavior of an evolutionary system. Banzhaf and Haken's [4], Hanson's [16], Kanter's [27], and Churchland's [6] papers describe connectionist models of learning; Greening's paper [15] discusses parallel simulated annealing techniques; Omohundro [41] examines geometric learning algorithms. Papers on the emergence of symbolic reasoning systems from subsymbolic components include Mitchell and Hofstadter [38] (models of analogy-making) and Harnad [17] (connectionism and the symbol-grounding problem).

Several papers describe emergent computations in different biological systems, ranging from the cortex to the cytoskeleton. Reeke and Sporns [45] discuss perceptual and motor systems. Two papers

(Baird [3] and Siegel [49]) focus on the cortex, Keeler's paper [29] examines cerebellar function, and George et al. [13] consider vision. Finally, Rasmussen et al. [43] present a connectionist model of the cytoskeleton.

Acknowledgements

I am grateful to Doyne Farmer, John Holland, Melanie Mitchell, and Quentin Stout for their careful reading of the manuscript and many helpful suggestions. Chris Langton and I have had many productive discussions of these ideas over the years.

References

[1] G.M. Amdahl, Validity of the single processor approach to achieving large-scale computing capabilities, AFIPS Conf. Proc. (1967) 483–485.
[2] R. Axelrod, An evolutionary approach to norms, Am. Political Sci. Rev. (1986) 80.
[3] B. Baird, Bifurcation and learning in oscillating neutral network models of cortex, Physica D 42 (1990) 365–384, these Proceedings.
[4] W. Banzhaf and H. Haken, An energy function for specialization, Physica D 42 (1990) 257–264, these Proceedings.
[5] A.W. Burks, ed., Essays on Cellular Automata (University of Illinois Press, Urbana, IL, 1970).
[6] P.M. Churchland, On the nature of explanation: a PDP approach, Physica D 42 (1990) 281–292, these Proceedings.
[7] E.F. Codd, Cellular Automata (Academic Press, New York, 1968).
[8] M. Compiani, D. Montanari and R. Serra, Learning and bucket brigade dynamics in classifier systems, Physica D 42 (1990) 202–212, these Proceedings.
[9] H.B. Curry and R. Feys, Combinatory Logic, Vol. I (North-Holland, Amsterdam, 1968).
[10] J.D. Farmer, A Rosetta Stone for connectionism, Physica D 42 (1990) 153–187, these Proceedings.
[11] J.D. Farmer, N.H. Packard and A.S. Perelson, The immune system, adaption, and machine learning, Physica D 22 (1986) 187–204.
[12] S. Forrest and J. Miller, Emergent behaviors of classifier systems, Physica D 42 (1990) 213–227, these Proceedings.
[13] J.S. George, C.J. Aine and E.R. Flynn, Neuromagnetic studies of human vision: noninvasive characterization of functional architecture, Physica D 42 (1990) 411–427, these Proceedings.
[14] D.E. Goldberg, Genetic Algorithms in Search Optimization, and Machine Learning (Addison–Wesley, Reading, MA, 1989).
[15] D.R. Greening, Parallel simulated annealing techniques, Physica D 42 (1990) 293–306, these Proceedings.
[16] S.J. Hanson, A stochastic version of the delta rule, Physica D 42 (1990) 265–272, these Proceedings.
[17] S. Harnad, The symbol grounding problem, Physica D 42 (1990) 335–346, these Proceedings.
[18] W.D. Hillis, Co-evolving parasites improve simulated evolution as an optimization procedure, Physica D 42 (1990) 228–234, these Proceedings.
[19] D.R. Hofstadter, Artificial intelligence: subcognition as computation, Technical Report 132, Indiana University, Bloomington, IN (1982).
[20] J.H. Holland, Adaption in Natural and Artificial Systems (University of Michigan Press, Ann Arbor, MI, 1975).
[21] J.H. Holland, Concerning the emergence of tag-mediated lookahead in classifier systems, Physica D 42 (1990) 188–201, these Proceedings.
[22] J.H. Holland, K.J. Holyoak, R.E. Nisbett and P. Thagard, Induction: Processes of Inference, Learning, and Discovery (MIT Press, Cambridge, MA, 1986).
[23] J.E. Hopcroft and J.D. Ullman, Introduction to Automata. Theory, Languages, and Computation (Addison–Wesley, Reading, MA, 1979).
[24] B.A. Huberman, The performance of cooperative processes, Physica D 42 (1990) 38–47, these Proceedings.
[25] T. Ikegami and K. Kaneko, Computer symbiosis – emergence of symbiotic behavior through evolution, Physica D 42 (1990) 235–243, these Proceedings.
[26] V. Jacobson, personal communication.
[27] I. Kanter, Synchronous or asynchronous parallel dynamics – Which is more different, Physica D 42 (199) 273–280, these Proceedings.
[28] S.A. Kauffman, Requirements for evolvability in complex systems: orderly dynamics and frozen components, Physica D 42 (1990) 135–152, these Proceedings.
[29] J.D. Keeler, A dynamical systems view of cerebellar function, Physica D 42 (1990) 396–410, these Proceedings.
[30] J.O. Kephart, T. Hogg and B.A. Huberman, Collective behavior of predictive agents, Physica D 42 (1990) 48–65, these Proceedings.
[31] J.E. Laird, A. Newell and P.S. Rosenbloom, Soar: an architecture for general intelligence, Artificial Intelligence 33 (1987) 64.
[32] C.G. Langton, Studying artificial life with cellular automata, Physica D 22 (1986) 120–149.
[33] C.G. Langton, ed., Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity (Addison–Wesley, Reading, MA, 1989).
[34] C.G. Langton, Computation at the edge of chaos: phase transitions and emergent computation, Physica D 42 (1990) 12–37, these Proceedings.
[35] E.N. Lorenz, Deterministic nonperiodic flow, J. Atmos. Sci. 20 (1963) 130–141.
[36] R. Machlin and Q.F. Stout, The complex behavior of simple machines, Physica D 42 (1990) 85–98, these Proceedings.

[37] R.A. Maxion, Toward diagnosis as an emergent behavior in a network ecosystem, Physica D 42 (1990) 66–84, these Proceedings.

[38] M. Mitchell and D.R. Hofstadter, The emergence of understanding in a computer model of concepts and analogy-making, Physica D 42 (1990) 322–334, these Proceedings.

[39] A. Newell and H.A. Simon, A program that simulates human thought, in: Computers and Thought, eds. E.A. Feigenbaum and J. Feldman (McGraw-Hill, New York, 1963) 279–296.

[40] G. Nicolis and I. Prigogine, Self-Organization in Nonequilibrium Systems (Wiley, New York, 1977).

[41] S.M. Omohundro, Geometric learning algorithms, Physica D 42 (1990) 307–321, these Proceedings.

[42] J. Palmore and C. Herring, Computer arithmetic, chaos and fractals, Physica D 42 (1990) 99–110, these Proceedings.

[43] S. Rasmussen, H. Karampurwala, R. Vaidyanath, K.S. Jensen and S. Hameroff, Computational connectionism with neutrons: a model of cytoskeletal automata subserving neural networks, Physica D 42 (1990) 428–449, these Proceedings.

[44] S. Rasmussen, C. Knudsen, R. Feldberg and M. Hindsholm, The Coreworld: emergence and evolution of cooperative structures in a computational chemistry, Physica D 42 (1990) 111–134, these Proceedings.

[45] G.N. Reeke Jr. and O. Sporns, Selectionist models of perceptual and motor systems and implications for functionalist theories of brain function, Physica D 42 (1990) 347–364, these Proceedings.

[46] D.E. Rumelhard, J. L. McClelland and the PDP Research Group, Parallel Distributed Processing: Explorations in the Microstructure of Cognition (MIT Press, Cambridge, MA, 1986).

[47] J.D. Schaffer, R.A. Caruana and L.J. Eshelman, Using genetic search to exploit the emergent behavior of neural networks, Physica D 42 (1990) 244–248, these Proceedings.

[48] R. Shaw, Strange attractors, chaotic behavior, and information flow, Z. Naturforsch. 36a (1981) 80–112.

[49] R.M. Siegel, Non-linear dynamical system theory and primary visual cortical processing, Physica D 42 (1990) 385–395, these Proceedings.

[50] H.A. Simon, The Sciences of the Artificial (MIT Press, Cambridge, MA, 1969).

[51] Q. Stout, personal communication.

[52] J.E. Stoy, Denitational Semantics: The Scott–Strachey Approach to Programming Language Theory (MIT Press, Cambridge, MA, 1977).

[53] E.O. Wilson, The Social Insects (Belknap/Harvard Univ. Press, Cambridge, MA, 1971).

[54] E.O. Wilson, Sociobiology (Belknap/Harvard Univ. Press, Cambridge, MA, 1975).

[55] S.W. Wilson, Perceptron redux: emergence of structure, Physica D 42 (1990) 249–256, these Proceedings.

[56] S. Wolfram, Universality and complexity in cellular automata, Physica D 10 (1984) 1–35.

[57] W.H. Zurek, Algorithmic randomness and physical entropy, Phys. Rev. A 40 (1989) 4731–4751.

[58] W.H. Zurek, Thermodynamic cost of computation, algorithm complexity and the information metric, Nature 341 (1989) 119–124.