NTNU
Det skapende universitet

Cellular Programming on Machines with Local Communication Networks

Torkel Berli

Master i datateknologi
Innlevert: februar 2017
Hovedveileder: Gunnar Tufte, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknologi og informatikk

Abstract

Efficient parallel computing is still a difficult goal to achieve, despite being of research interest for more than half a century. For the past decade, however, the topic has become increasingly more relevant as a result of emerging obstacles to improvements in the complex von Neumann-based, single-CPU computer. Using many processors to cooperate on a single task concurrently promises benefits to energy efficiency and computational performance. Programming such many-core systems, however, is challenging.

The primary obstacle for efficiently utilizing a parallel processor is the complexity of programming such parallel machines. It has been suggested that humans lack the ability to manually solve this problem, and that it should be automated. One possibility is to use methods inspired by artificial life research. By way of self-organization and an evolutionary search, efficient behavior in a parallel machine may emerge. A prerequisite for such a process is a framework of high evolutionary adaptability in which to structure the program.

Evolutionary processes have previously been used with cellular automata to let many simple processing cells cooperate to solve a problem. In this project, we apply principles from cellular automata research to program a locally connected, homogeneous multi-core processor. We propose a framework inspired by both cellular programming and genetic programming. The framework uses an evolutionary process to automatically structure software among the processor cores. With three example problems, we demonstrate the framework's ability to develop and spread useful behavior among the processor cores over several generations.

The evolutionary process evaluates correctness of behavior using local information only, and this is the main challenge for the framework. The problem of writing local fitness functions that produce a desired behavior on a global level is difficult and requires more work. While the framework shows desired behavior over evolutionary processes, it is difficult to conclude its usefulness before further progress in problem representation and fitness evaluation is made.

Sammendrag

Effektiv parallell beregning er fortsatt vanskelig å oppnå, til tross for at forskningsfeltet er mer enn 50 år gammelt. Det siste tiåret, derimot, har feltet blitt mer relevant igjen på grunn av en redusert evne til å fortsette forbedringer i kompliserte, von Neuman-baserte enkjærners prosessorer. Ved å la mange prosessorkjærner samarbeide på en og samme oppgave, er det sannsynlig at vi kan øke både energieffektivitet og beregningshastiget. Men å programmere slike parallelle systemer er en utfordring.

Hovedutfordringen i å kunne effektivt utnytte evnen til en parallell prosessor er å overkomme kompleksiteten i programmene for slike parallelle maskiner. Det har vært foreslått at mennesker kanskje ikke evner å løse denne utfordringen manuelt, og at programmeringen må automatiseres. En mulighet er å benytte metoder inspirert av forskning innen kunstig liv. Ved hjelp av selv-organisering og evolusjonære metoder, så kan program som evner å utnytte parallellitet på en effektiv måte oppstå. En forutsetning for en slik prosess er et godt rammeverk som legger til rette for evolusjonære prosesser.

Evolusjonære prosesser har tidligere vært brukt sammen med cellulære automater til å få mange enkle enheter til å samarbeide for å løse problemer. I dette prosjektet bruker vi prinsipper lært fra forskning på cellulære automater til å programmere lokalt koblede mangekjærners prosessorer. Vi presenterer et rammeverk inspirert av cellulær programmmering og genetisk programmeringsmetoder. Dette rammeverket bruker en evolusjonær metode for å automatisk generere struktur i programvare for mangekjærners prosessorer. Gjennom tre forskjellige problemer demonstrerer vi rammeverkets evne til å utvikle og spre nyttig oppførsel mellom prosessorkjærnene over mange generasjoner.

Den evolusjonære metoden evaluerer om oppførsel er riktig kun basert på lokal informasjon. Det er hovedutfordringen med dette rammeverket. Problemet ligger i å utvikle lokale evalueringsmetoder som forårsaker ønsket oppførsel på et globalt nivå. Det kreves videre forskningsarbeid for å nærme seg en løsning på denne utfordringen. Selv om rammeverket viser ønsket oppførsel ved bruk av evolusjonære metoder, så er det vanskelig å konkludere i hvilken grad rammeverket er godt før forskningen har gjort fremskritt innen lokale evalueringsmetoder og representasjon av problemfunkjsoner.

Preface

This Master's Thesis is the final deliverable of the Computer Science program at the Department of Informatics, Norwegian University of Science and Technology. The research was conducted by Torkel Berli over the last semester of the study program, and was completed in February 2017. Gunnar Tufte at the Department of Informatics, NTNU served as supervisor.

Acknowledgements

I would like to thank *Gunnar Tufte* for being my supervisor for this project. Gunnar has helped me with forming the goals and direction of this research project, and has helped me navigate intertwined and at times confusing concepts of artificial life and computer architecture. I have benefited from his experience and knowledge for this project as well as previous projects at NTNU.

I would also like to thank *Kristian Drsshaug* at the Faculty of Information Technology and Electrical Engineering, NTNU. His support and counsel, and administrative aid, was important to me for my final year at NTNU. I was always welcome to have a chat if I needed to, and for that I am grateful.

Lastly, my family deserves mentioning, for they have been critical to my success for the duration of my studies, particularly for this final project. Mom and Dad have looked forward when I have looked back, looked up when I have looked down. I am grateful for their unrelenting support. I am thankful also for the time spent with Bjørnar at the soccer pitch, at Lerkendal, and at the dinner table.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter explains the motivation for and aim of the research topic presented in this paper. It describes the goal of the project, and how work was conducted to reach that goal. It then lays out the structure of the content of this paper.

1.1 Background and Motivation

In computer science, finding novel ways of performing computation is an ongoing research goal. This is motivated by the difficulties of further improving the computational potential of traditional processors. Traditionally, computers execute a program consisting of a series of logical operations decided manually by a human programmer, and the program arrives at a deterministic answer. This method is relatively easy to understand and predict for humans. We are, however, experiencing stagnation in the technological advances that are possible with such computing models. The causes of this stagnation includes problems with power dissipation in processors, an increasing effort required to move data around, and difficulty in manually exploiting computational parallelism [**?**].

In a paper from 2006 [5], several professors from University of California at Berkeley illustrated how conventional wisdoms of computing research was changing. Processor performance had increased by 52% per year between 1978 and 2002, but by less than 20% per year in the following 3 years. Three primary obstacles halted the rapid progression previously experienced. First, as processors get faster, they produce exponentially more heat. Processors hit what is referred to as the *Power wall*, where amount of power spent became the primary limitation of the speed and size of processors. Second, the speed of processing instructions started outpacing the time to access memory. referred to as the *Memory wall*. In other words, moving data around quickly enough became an obstacle for speeding up computation. Third, architecture and compiler innovation had nearly exhausted the potential for exploiting instruction-level parallelism (ILP) in sequential programs. This is known as the *ILP wall*.

The paper from Berkeley is titled *The Landscape of Parallel Computing Research*, and parallel computing is seen as the solution to the obstacles above. The size and cost of transistors has decreased (as predicted by Moore's infamous paper of 1965 [17]) to the point where adding more elements to an integrated circuit (IC) is essentially free. Thus, in an ideal world, we could sustain the previous, enormous growth in computational power by using an ever increasing number of slower, less power-hungry processing elements. Taking advantage of many processors is a difficult task, however. To quote the Berkeley professors: "This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures."

Attempts at creating effective parallel processors started already in modern computer's infancy, but the effort has taken much longer than at first anticipated. In a 1996 article [11], Michael J. Flynn[1] takes a retrospective look at this effort since the 1960's. He notes that the difficulty in improving performance through parallel processing was greatly underestimated, and that our mathematical way of representing computing problems may be a fundamental obstacle. He calls for researchers to approach parallel processing by representing problems in cellular form; an inherently parallel representation difficult to

[1] Michael J. Flynn is known for his classification of computer architectures from 1966, "Flynn's taxonomy", which is based on the number of parallel instruction and data streams that the architecture supports.

understand for the human mind but tailored to the parallel machine.

As the number of independent processors in a system increases, human ability to manually write efficient programs for it decreases. In an effort efficiently utilize the computational resources of vast parallel systems, we must look to automate the problem of programming such systems. This is a logical conclusion of Michael Conrad's Trade-off Principle on programmability of systems [6]. Conrad suggests that the concepts of self-organization and evolution may be exploited in vast parallel systems to create powerful programs that are too complex for a human to understand.

Structuring a program by way of self-organization and an evolutionary search represents the bottom-up approach of artificial life research. Inspired by computation-like phenomena in biology, such as the collective behavior of ant swarms and neuron interaction in the brain, the goal is for behavior to *emerge* out of many simple, locally interacting parts.

In order to utilize evolutionary search to program a system, the system's framework must be structured in a way that facilitates this process. One important feature of the framework is that structural changes to the program may result in small changes to the outcome, without compromising successful execution. This does generally not hold for traditional programming frameworks. An evolvable system must also be flexible enough such that a series of structural changes may result in a wide variety of behaviors. Conrad argues that cellular models provide the required features of an evolvable system, and as such, have a higher ceiling for computational efficiency than manually programmable systems.

1.2 Research Goals

For this project, we draw inspiration from previous work within the fields of artificial life, evolutionary algorithms and cellular automata. We look to program a parallel processor using techniques developed for cellular automata.

The parallel processor we work with is called Epiphany, and its processor cores are connected with a local communication network. The framework we propose for structuring a program on Epiphany is a combination of cellular programming and genetic programming. It mirrors the evolutionary process of cellular programming, suitable for the topology of the interconnections of the processor array. But the genetic code for cell behavior follows the genetic programming technique, suitable for a cell with full processor capabilities.

Our immediate goals of the project are:

- Demonstrate that cellular programming may be a framework for structuring software on machines with local communication networks.
- Run the resulting software on the Epiphany parallel processor.

The Epiphany parallel processor has an interesting grid-like organization of CPU cores, and we look to explore how cellular computing may benefit from such an architecture. At the same time, we are interested in how non-traditional computing frameworks, such as cellular programming, contribute to diversity in modern computer architectures.

We note that this effort is a microscopic step towards the long-term goals of these research themes, and the progress made in this project is of little immediate use outside of research purposes. Long-term goals include the ability to efficiently utilize massively parallel machines, to investigate alternative ways of organizing a computer, and to better understand the computational dynamics of many biological phenomena.

1.3 Thesis Structure

This thesis' content is divided into six chapters.

- **Chapter 2** presents background theory on topics relevant for this thesis, including modern parallel processor architectures, artficial life, and cellular automata.
- **Chapter 3** presents the hardware platform on which we will implement the programming framework of chapter 4.
- **Chapter 4** describes the programming framework we propose in this thesis, and show the results of its application to test problems.
- **Chapter 5** outlines an implementation of the parallel program on our hardware platform.
- **Chapter 6** discusses the results of our findings, outlines important topics of the framework that needs further work, and gives a conclusion for the project.

Chapter 2

Background Theory

This chapter covers background theory for the work presented in this paper. First, we introduce the academic field of artificial life and important concepts like emergence, self-organization, and evolution. Next, we describe the paradigm of cellular computing and the cellular automata model in particular, which is commonly used to model artificial life phenomena. Last, we look at methods for developing useful structures in cellular automata.

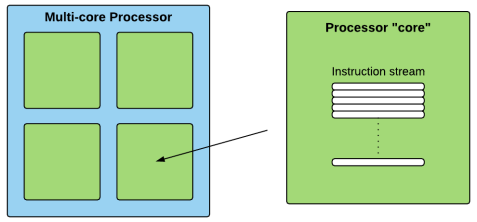


Figure 2.1: Multi-core Processor.

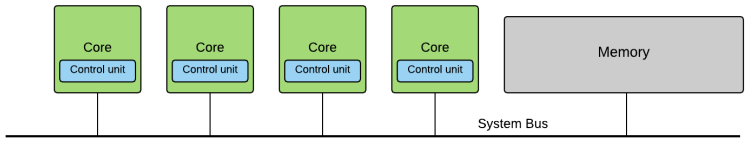


Figure 2.2: Multi-core MIMD system.

2.1 Modern Parallel Processors

The concept of executing a series of instructions by reading and manipulating registers, known as the von Neumann model, is common to the way all processors perform computation. But the organization surrounding this execution loop may vary. We can categorize processor architectures by the number and the variety of processing units they use, and how these units are coordinated.

A processor system with several processing units, or ”cores”, is called a multi-core processor. Each core independently executes a stream of instructions, although some or all of the cores may share instruction stream. Figure 2.1 serves to clarify these terms. Typically, the set of cores in a multi-core processor are physically integrated on a single integrated circuit, known as a chip multiprocessor (CMP).

Computing systems employ more than one processor core for reasons of computational performance or energy efficiency. Cores may execute separate programs or they may collaborate on a single problem. In either case, as the number of processors increase, so does the difficulty of utilizing the available resources in an efficient manner [6].

2.1.1 Homogeneous Multi-core Processors

Homogeneous multi-core processors are systems in which all cores are equal. This implies that any workload may be scheduled to any of the system’s cores, and each core must have the resources to carry out any task. This generalist approach makes scheduling tasks easier, but cores will often be inefficient at executing its assigned tasks.

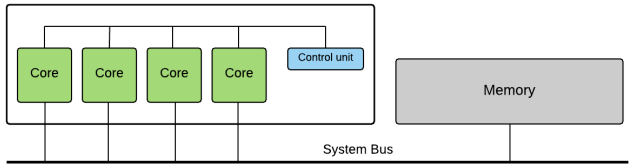


Figure 2.3: Multi-core SIMD system.

Modern off-the-shelf CPUs typically have from two to eight indentical cores, organized as shown in Figure 2.2. The system's interconnection network may vary, but all cores have access to the same memory. A control unit in each core handles the instruction stream. This is categorized as a multiple instruction, multiple data (MIMD) system by Flynn's taxonomy.

Some homogeneous systems group several cores together to share instruction stream, as in Figure 2.3. This is categorized as single instruction, multiple data (SIMD), and is a typical configuration for graphics processors. A SIMD processor is designed for highly specific, parallelizable problems such as pixel coloring and certain scientific computing problems.

2.1.2 Heterogeneous Multi-core Processors

Heterogeneous multi-core processors include different types of cores on one system in order to utilize resources more effectively. A task will be scheduled to a specialized processing unit, according to the type of computation involved. This specialist approach lets the system spend less time and energy for a given task, but scheduling tasks to each unit is more complicated than for homogeneous systems.

One example of a modern heterogeneous processor is ARM's big.LITTLE [4], designed for use in mobile devices. It includes a set of high-performance, energy intensive cores alongside a set of slower, energy efficient cores. Both types run the same programs, and a scheduler selects which types of cores to use based on the current performance demands. This configuration lets big.LITTLE switch between energy efficiency and high performance on a whim.

2.1.3 Utilizing Multi-core Processors Efficiently

For some highly parallelizable problems, modern parallel machines are very efficient. Some SIMD machines, like graphics processors, are tailored to a narrow use-case such that it has precisely the resources required for its expected tasks. The regularity of the SIMD machine matches the regularity of the parallel algorithms. For general purposes, however, this type of architecture is limited in its ability to employ its resources efficiently. Less specialized SIMD architectures (as compared to graphics processors) sacrifice parallelism for more general purpose relevance.

The graphics processor is an example of a parallel machine for which the programmer is responsible for proper utilization of resources. Parallelizable sections of the program are manually specified to run in a SIMD manner, often through a parallel programming framework such as OpenCL. The responsibility of parallelization falls on the programmer in many other situations also, such as for heterogeneous multi-core processors with more than one ISA. Processors with different ISA may not run the same programs, so the programmer must choose which processor to write software for.

Note also that parallel algorithms for programmable systems are constrained by the need to coordinate the parallel execution in non-parallelizable portions of the program (see Amdahl's Law). This restricts their potential for efficiency as the number of parallel elements scale up.

For systems in which the programmer is responsible for parallelization, a theory relating the *programmability* of the system to its *potential computational efficieny* claims that these two properties are bound by a trade-off principle [6]. In other words, a programmable, vastly parallel system cannot achieve high computational efficiency. In order to achieve efficiency, the vastly parallel system must be *evolutionary adaptable*. And its programs must self-organize through a variation and selection process, according to the theory.

2.2 Artificial Life

The academic field of artificial life studies systems and processes that exist in our natural environment using simplified models. In contrast to traditional biology which primarily studies such phenomena through empirical investigation, artificial life seeks to recreate the phenomena and understand them at a conceptual level. Through simulation of a variety of life-like models, this study may advance our knowledge of the logical principles of life, separated from the peculiarities of life in our natural environment. Methods of modelling include computer simulations, robotics, and biochemistry.

How the vast number of non-living molecules that make up our body manages to collectively create a living, self-aware human being is not well understood. This mystery involves several yet unanswered questions in biology, such as how multicellular replication evolved, and what are the requirements for conciousness. These, among many others, are yet unanswered questions that are likely to benefit from analysis of artificially living systems [25]. But research in artificial life contributes not only to answer open questions in biology, it may also improve our ability to perform computation using biology-inspired models.

2.2.1 Emergence

Advances in computer technologies opened the door for simulation of larger and more powerful models that can capture the complex interactions of natural life systems. Natural life systems exist both at microscopic scales as with cell biology, and at large scales such

as populations in an ecosystem. Common to these systems is that order *emerges* out of interactions between a vast number of individual elements. We can define this phenomenon as follows:

> Emergence is a dynamical process in which global behavior or structure arises from local interactions between individual parts in a system.

Emergent properties develop over time, at a macro-level of a dynamical system. These properties cannot be described by the behavior of local parts, i.e. at the micro-level, without also considering the context with which they interact.

As an example, traffic congestion can be considered an emergent property of the collection of vehicles in an area with high traffic density [16]. While the drivers would like to avoid congestion, their behaviors inevitably influence each other. The phenomenon emerges in the interaction between the drivers as they brake, accelerate, and manouver to avoid collision and navigate the road network. Congestion forms over time, and may spread across the road network or die out.

Emergent behavior is not directed by a central source of control, and no single element can be critical to its success. The individual behaviors of all elements of the system gives rise to its collective behavior, and as such, the system is *robust* in the presence of failure or damage. While failure in parts of the system may reduce performance of the emergent behavior, the distributed responsibility of control ensures graceful degradation. In the example of traffic congestion, removing any one vehicle does not significantly impact the emergent behavior.

The concept of emergence appears extensively in fields such as artificial life and evolutionary biology [9]. It is also found in other disiplinces, such as economics, chemistry, and ecology. Despite an increasing interest from multiple disciplines for the past 20-30 years, the theory of how emergent behaviors form is not well understood [7].

2.2.2 Self-organization

Self-organization is another property of dynamical systems, and it often occurs in combination with emergence. It may be defined as follows [9]:

> Self-organization is a dynamical and adaptive process where systems aquire and maintain structure themselves, without external control.

Here, 'structure' is some form of order, whether it is spatial, temporal, or functional in nature. A self-organizing system is able to autonomously increase its order over time in a way that promotes a certain function or property. That it does so 'without external control' does not mean without external input, but rather without direct control from an external agent.

A self-organizing system is *adaptive*, able to change or maintain its structure in the face of disturbances or changing contexts. This implies that the system cannot exhibit too much order. With an exessively high degree of order, the system's structure will be too static. An adaptive system must also consider a variety of different behaviors and choose which

path to take. With a low degree of order, the system will be uncontrollable. To be able to respond meaningfully to variations in context, a self-organizing system must balance on the edge of order and chaos [12][15].

2.2.3 Evolution

Evolution is the process by which the genetic codes of a population of organisms change and adapt over successive generations. In biology, evolution by natural selection is a source of a species' adaptation to the environment, as well as the formation of new species. In computer science, *evolutionary algorithms* are used to apply the principles of evolution to solve optimization problems.

Evolution is driven by two main processes: *variation* and *selection*. Variation takes place when a new specimen, with its unique genetic code, is introduced to the population. For example, variation in the human species takes place through sexual reproduction, by forming the genetic code of the child from a combination of the parents' genetic code. Mutation in genes is also a source of variation, and higher *mutation rate* results in more variation.

Selection is the process by which an evolving population adapts to its environment. It takes place when some specimen are more than average likely to reproduce and pass on its genes to the next generation, causing those genes to be selected for. Natural selection is based on the well-known principle of "survival of the fittest". *Fitness* is the term used to describe how good a particular specimen is, i.e. how well-adapted it is to its environment.

Evolutionary algorithms work by applying variation and selection to a population of potential solutions, or "artificial organisms". The goal is to let the population of solutions improve over many generations to produce better and better results. Fitness, in this case, is a score based on how well suited a solution is to solving the problem at hand. Selection is achieved by prioritizing high-fitness solutions for recombination and survival into the next generation.

Each solution or organism, also known as *phenotype*, in an evolutionary algorithm is a product of a form of genetic code, or *genotype*. The genotype may be represented by a simple bit string. Operators for performing variation and selection in this genetic code varies between different evolutionary algorithms. One of the most popular evolutionary algorithms is the genetic algorithm (GA), illustrated in Figure 2.4 to show the typical process of artificial evolution.

The GA starts with a more or less random initial population. Individuals in this generation are evaluated by a *fitness function* to receive a numerical score for its fitness. Next it generates a full set of new individuals to form the next generation. A new individual is created by choosing two parent individuals and combining sections of each of their genotypes into a new offspring genotype, a process known as *sexual recombination*. Parents are selected probabilistically based on their fitness score. Each new genotype may also, with low probability, be selected for *mutation* of one or more genes. The complete set of new offspring individuals form a new generation, and the process is repeated.

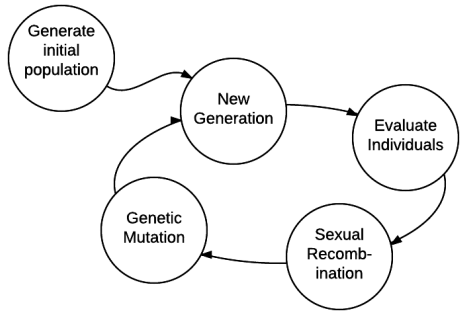


Figure 2.4: Genetic algorithm procedure.

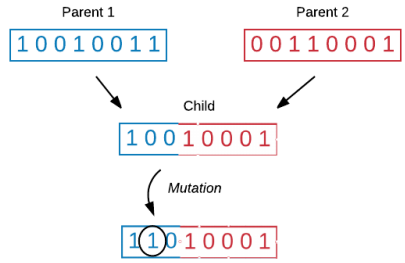
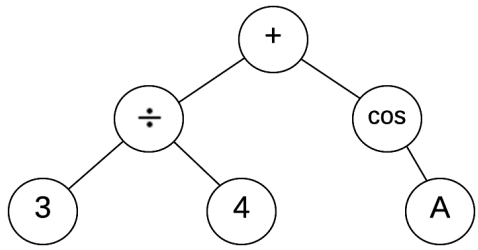


Figure 2.5: Sexual recombination and mutation.



$$\frac{3}{4} + \cos(A)$$

Figure 2.6: Example mathematical tree structure, a representation used in genetic programming.

Evolutionary algorithms are particularly useful for problems for which it is difficult to engineer a solution, but easy to evaluate if a potential solution is good or not. Programming highly parallel systems is one such example [6].

Evolutionary algorithms are also a tool for investigating the underlying dynamics of the evolutionary process. Philosophically, evolution relates to 'why' certain phenomena or traits appear, in contrast to emergence and self-organization which relates to the 'how'. For example, evolution may explain why humans have eyes, while emergence and self-organization may explain how our visual system works.

2.2.4 Genetic Programming

Genetic programming is a technique used with evolutionary processes to evolve a function in the form of a mathematical tree structure. Its fitness is based on the function's ability to perform a desired computation. An example tree structure and its corresponding function is shown in Figure 2.6. Mapping genotypes to mathematical tree structures in a way that facilitates evolution is challenging, and subject to ongoing research.

2.2.5 Bio-inspired Computation

Artificial life research is relevant for computer science in attempting to discover and understand novel ways with which to perform computation [8]. Many systems we find in our natural environment process information in a parallel and distributed manner. We refer to it as *emergent computation* when systems of locally interacting parts process information in a global manner (ref. Section 2.2.1). Examples of this include collective food gathering [18] and nest-building [14] in ant societies, and processing of nerve impulses by neurons in the brain [21].

Computational systems in biology consist of parts that may fail, or that may be non-deterministic. It performs reliable computation with unreliable parts, and has properties

such as adaptation and robustness in the face of changing circumstances or failure. This is a key difference between biological systems and a bio-inspired machine. A machine consists of reliable parts.

2.3 Cellular Automata

Since the 1990's, researchers have explored cellular models for their potential for performing computation. Cellular models rely on a high degree of parallelism, and researchers hope that such models have computational potential that exceeds the limitations of traditional, sequential methods [22]. The most influential cellular model is the cellular automaton (CA).

2.3.1 Cellular Computing

The computational paradigm of cellular computing is based on three main principles: simplicity, vast parallelism, and locality [22]. It utilizes the computational potential of a vast network of simplified processors (cells) that can only interract locally, i.e. with a relatively small number of other nearby cells. The set of other cells that a cell can communicate with is referred to as its *neighborhood*.

The lack of a central source of control, guidance, or communication is the key factor that separates cellular computing from traditional computing. Each processing element communicates with a constant number of other units, regardless of how many processing elements the system has in total. This is a great advantage over traditional, centrally directed computing models, as it lets cellular machines scale without increasing communication overhead. Furthermore, parallel computation in traditional computing models is based on solving independent sub-problems. Thus, the potential parallel speedup such models can achieve is bound by the portion of parallelizable work (see "Amdahl's Law"). The fully distributed models of cellular computing are not bound by such restrictions, primarily because cells do not coordinate at a single processing unit.

2.3.2 The Cellular Automaton Model

Cellular automata are a special case of cellular models. Its cells are arranged in a fixed array of one, two or three dimensions forming a line, rectangle, or a box respectively. The state of each cell has a discrete value, and all cells update their state synchronously, over discrete time steps.

Figure 2.7 illustrates a 4x4 CA in 2D with a neighborhood of 5 cells (including itself). This neighborhood format is called the *von Neumann neighborhood* of a two dimensional CA. Similarly, the von Neumann neighborhood of a one dimensional CA includes 3 cells: Center, West, and East.

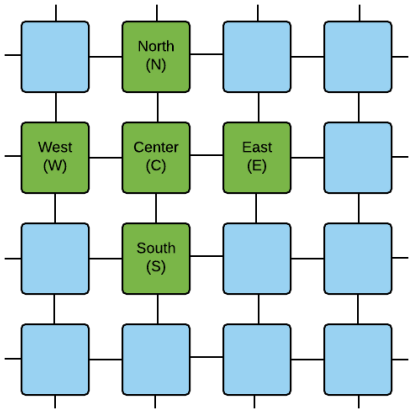


Figure 2.7: Two-dimensional 4x4 cellular automata showing the von Neumann neighborhood of the center cell.

Table 2.1: An example 3-neighbor transition function of a one dimensional CA.

Rule Table

C	W	E	Next state
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Bit string	Integer
0 0 1 1 1 0 1 1	59

Each cell takes as input the current state of the cells in its neighborhood, and at each time step updates its own state according to a *transition function*. This transition function is traditionally implemented as a look-up table (LUT), defining the cell's next state for each possible combination of inputs. Table 4.2 shows an example rule table for a binary cell with a neighborhood of 3 cells. It has 2 possible states and 3 inputs, and therefore 2 entries for which a "next state" value is defined. These values may be represented in compressed form as a simple bit string, as shown.

The cells of a CA may have an arbitrary number of possible states. However, binary cells are the norm, as in Table 4.2. Increasing the granularity of the state space or increasing the neighborhood of each cell leads to extreme increases in the total number of possible input combinations. Since CA cells often use LUTs for implementation, and the size of each LUT scales proportionally with input combinations, it is often practical to use simple CAs for experiments. Furthermore, the type of dynamical behavior found in CAs with more than two states per cell does not seem to be qualitatively different from the behavior of binary CAs [27].

A CA in which all cells are defined by the same transition function is referred to as a *uniform* CA. A *non-uniform* CA on the other hand has independent transition functions in each of its cells. Sometimes a third category, *quasi-uniform*, is used. The cells of a quasi-uniform CA are predominantly of one or two main transition functions. but the array may also contain a few 'gaps', small collections of cells that use other transition functions. Non-uniform CAs are discussed in section 2.4 below.

Note that, in practice, the CA is of finite size. Instead of specifying a static input from the border of the cell array (e.g. "West" input for the west-most row of cells), the CA will typically wrap around, connecting its edges. A CA in one dimension will therefore form a circle, while a CA in two dimensions will form a torus.

2.3.3 Signals as Emergent Behavior in CA

Research from [13] on emergent behavior in CAs illustrate the ability of locally interacting cells to develop collective behavior at a larger scale. When drawing the behavior of a one-dimensional CA over time, as seen in Figure 2.8, we can identify different domains in its behavior. The left picture in the figure shows how the pattern of states in the CA form idenfifiable domains, and the right picture draws the border between these domains.

Black domains identify areas of black cell dominance initially, and white domains identify white cell dominance. The borders between these domains can be seen as signals propagating over the cell array, as a form of communication. When these communication signals connect, the cell array makes a decision of what domain to proceed with. This domain and signal behavior is a form of emergent behavior in the CA.

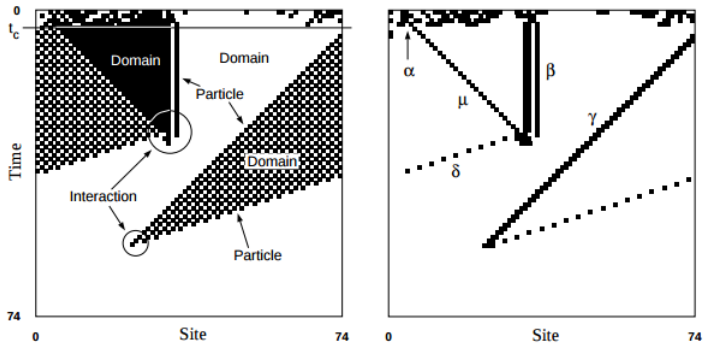


Figure 2.8: Emergent behavior in cellular automata. (Reprinted from [13].)

2.4 Development of Structure in CA

So far, our discussion of CAs has been limited to CAs with uniform rulesets. In this section, we discuss non-uniform CAs and ways of handling their more complicated rulesets.

2.4.1 Uniform vs Non-uniform Rulesets

Uniform CAs are easily identified by their rule. The length of the decoded rule may vary depending on neighborhood size, but it is nonetheless relatively easy to read, modify, and store. Rule 59 from Table 4.2, for example, only consists of 8 bits, while a two-dimensional CA with von Neumann neighborhood will have a 32-bit rule. Because of the easy representation, applying a genetic algorithm to a population of uniform CAs is quite accessible.

A non-uniform CA, however, must be described by all the individual rules in the array, one rule for each cell. We can refer to the set of rules and their distribution across the cells as the *structure* of the CA. The process of generating a structure that performs some function is the equivalent of ”programming” the non-uniform CA.

If we attempt to apply a genetic algorithm to the direct representation of an entire non-uniform CA, the search space becomes undesirably large. Furthermore, a potential solution of the genetic algorithm, the resulting structure, will be specific to one particular size and shape.

Instead of treating the entire non-uniform CA as one artificial organism directly, it is possible to organize the problem in a way that dynamically generates structure in the CA. Two such methods are covered in the sections below. The first method is a compact representation of how the structure can develop over time, inspired by multicellular organisms in biology. The second method treats each cell as a separate adaptive organism.

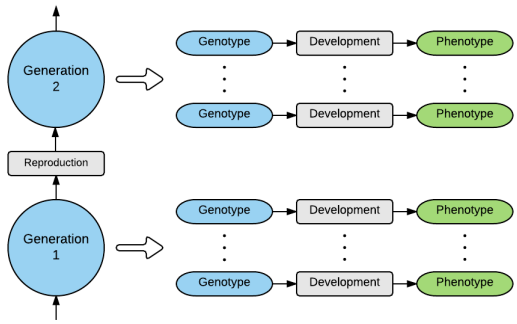


Figure 2.9: Diagram showing the relationship between development and evolution. (Reimagined from [24], Fig. 5).

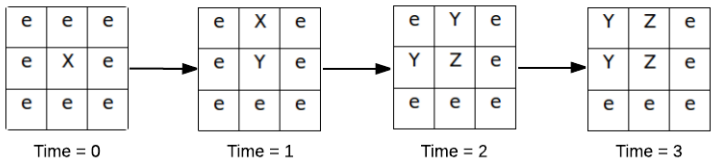


Figure 2.10: Artificial development of structure in a cellular automata.

2.4.2 Artificial Development

In traditional evolutionary algorithms, the genotype of an artifical organism is a string of bits that map directly into its phenotype. In biology, however, a cell's DNA does not structurally describe the organism. Instead, developmental processes in each cell use information in the DNA to grow a multicellular organism from a fertilized egg (which is simply one large cell). Modelling this aspect of biology in computing systems is known as artificial development [26]. Figure 2.9 illustrates the relationship between development and evolution.

The process of artificial development in a CA is shown in Figure 2.10. The figure shows four timesteps of a CA with uniform rulesets, but with four possible state values: *X, Y, Z*, and *e*. Each of these states represent a transition function for a non-uniform, binary CA. The distribution of the four states is the equivalent of the *structure* in a non-uniform CA.

In the figure, *e* represents an empty transition function, i.e. one that does not change its state. The first time step includes an *X* only, and the development process is defined by a set of rules that govern how this *X* seed should develop. With each time step of this development process, structure for the non-uniform CA emerges.

Mapping genotype to phenotype in an developmental model serves as an indirect, com-

pact way of describing a potentially large and complex structure. This property is helpful for successful evolution because small changes to the genotype will have the potential to produce sufficiently varied structures [10].

2.4.3 Cellular Programming

Cellular programming was introduced by Sipper as an evolutionary algorithm with which to program non-uniform CAs [23]. Instead of treating each CA as an artificial organism to evolve, it treats each individual cell as a separate artificial organism. The population of organisms in the evolutionary process is simply made up of all the cells of a single CA. An interesting property of cellular programming is that well-performing cells tend to spread their genes over time. In other words, structure develops in the CA dynamically. This is property is shared with artificial development.

Algorithm 1 shows pseudo-code for the cellular programming process, as given by Sipper. Cells initially receive a random genotype. The CA is then given an initial state configuration and set to run a set number of time steps. When time steps are completed, the performance of each cell is evaluated based on the fitness function. Several more initial configurations may be tested, with a new fitness evaluation each time.

The next step is to perform evolutionary operations on the population of cells, including sexual recombination and mutation to form a variety of behaviors for the next generation. The new generation is then evaluated in a similar manner, over several initial configurations. Over time, through many generations, a structure of high-fitness cells emerges in the CA.

2.4.4 Cellular Programming Example: Majority Problem

Sipper solved, in [23], the majority problem in one-dimensional CAs using cellular programming. The CA operates correctly if it can settle on an all-black or all-white state, depending on which cell type is in the majority at the initial condition. A graph of one of his solutions is shown in Figure 2.11. The squares show the state of the one-dimensional CA over time for two different initial conditions, one majority white and one majority black.

The lines below the squares have different colors for different sections of genotypes. There we can identify both growth and differentiation in gene selection. The rule space is semi-uniform, with all but a few cells having the same rules. The small sections of different rules are critical for the correct behavior of the CA, however. We can see how the uncommon rules help break up the emergent communication signals, and essentially make a decision of whether black is dominant or white is dominant.

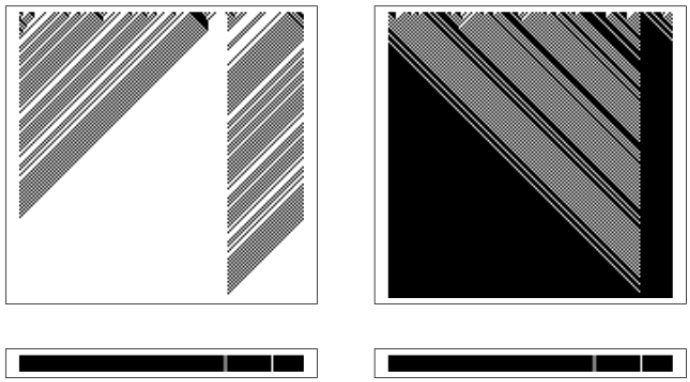


Figure 2.11: Majority problem solved using cellular programming. (Reprinted from [23].)

Algorithm 1 Pseudo-code of the cellular programming algorithm. (As given in [23])

for each cell i in CA **do in parallel**
    initialize rule table of cell i
    $f_i = 0$ { fitness value }
end parallel for
$c = 0$ { initial configurations counter }
while not done **do**
    generate a random initial configuration
    run CA on initial configuration for M time steps
    for each cell i **do in parallel**
        if cell i is in the correct final state **then**
            $f_i = f_i + 1$
        end if
    end parallel for
    $c = c + 1$
    if c mod $C = 0$ **then** { evolve every C configurations }
        for each cell i **do in parallel**
            compute $nf_i(c)$ { number of fitter neighbors }
            if $nf_i(c) = 0$ **then**
                rule i is left unchanged
            else if $nf_i(c) = 1$ **then**
                replace rule i with the fitter neighboring rule, followed by mutation
            else if $nf_i(c) = 2$ **then**
                replace rule i with the crossover of the two fitter neighboring rules, followed by mutation
            else if $nf_i(c) > 2$ **then**
                replace rule i with the crossover of two randomly chosen fitter neighboring rules, followed by mutation (this case can occur if the cellular neighborhood includes more than two cells)
            end if
            $f_i = 0$
        end parallel for
    end if
end while

Chapter 3

Hardware Platform: Epiphany-III

This chapter introduces hardware platform that we use for the experiments in this project. The processor for which we apply the cellular programming algorithm is a parallel processor architecture called Epiphany. We present the architecture in a top-down, informal manner and focus on the features that make it suitable for cellular programming, such as its manycore approach and local interconnect network.

3.1 Introduction

3.1.1 Brief History

The Epiphany architecture was invented in 2008 by Andreas Olofsson, with the goal of being a high-performance energy-efficient manycore architecture for use in real-time embedded systems [20]. Outside of performance and energy constraints, Olofsson wanted his architecture to scale to ”thousands of cores”, be easy to program, and require only a small team of engineers to implement. He designed each processor core as simple as possible, and connected them with a 2D mesh network.

Olofsson founded Adapteva to attempt to bring to life this architecture as a general purpose processor. The first product based on the Epiphany architecture was a 16-core System-on-Chip released in 2011 (”Epiphany-III”). Later that year, Adapteva completed and produced in low volume a 64-core version (”Epiphany-IV”), and built the first prototype of a 1024-core version. In 2016 due to funding from Defense Advanced Research Projects Agency (DARPA), Adapteva was able to successfully produce its latest version, Epiphany-V, which consists of 1024 cores and is scaled up from 32-bit to 64-bit [19].

The 16-core Epiphany-III shares a lot of its history and success with a small single-board computer called Parallella, which we will look at in Section 3.3.

3.1.2 Motivation

The Epiphany processor is used in this project for its grid based, locally connected, many-core architecture, and its accessibility in terms of hardware requirements and programmability. The layout of its cores and interconnection is structured much like a 2D CA with a von Neumann neighborhood.

For traditional, binary CAs, each cell is generally implemented using a simple rule table. In contrast, each ”cell” on Epiphany has much more powerful computational ability. For the cellular programming approach (Section 2.4.3), cells on Epiphany are therefore able to carry out the evolutionary operations internally, in parallel with each other. Furthermore, it lets us explore CA capabilities when cell behavior is not limited to binary values and a look-up-table.

3.2 Epiphany Architecture

This section describes the three core components of the Epiphany architecture: its memory system, CPU, and interconnect network. The official architecture reference [2] is the primary source for the information in this section, and can be referenced for further details about Epiphany.

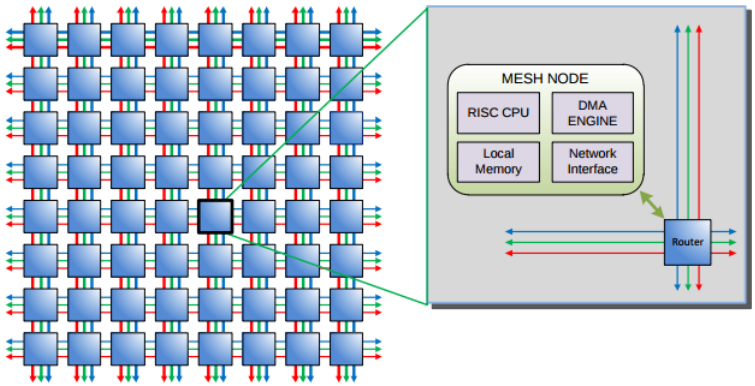


Figure 3.1: Overview of the Epiphany Architecture. (Reprinted from [2].)

3.2.1 Overview

Epiphany is a homogeneous multi-core processor consisting of a two-dimensional grid of computing nodes, as shown in Figure 3.1. The grid structure is easily scalable; the 32-bit system used by Parallella can scale to 4096 nodes on a single chip, limited by the number of memory address bits it uses.

Nodes contain a floating-point RISC CPU named *eCore* and some local SRAM. Each node is connected to its four closest neighbors (north, south, east, and west) by an efficient on-chip communication network named *eMesh*. In addition, all nodes share a single bus for accessing off-chip memory. Communication is handled by a DMA module and a network interface module included in each mesh node.

Some of the primary goals for the Epiphany architecture were energy-efficiency, floating-point performance, scalability, and ease of programmability. These goals strongly influenced the design decisions of its components. Features not included in the design were left out primarily to save energy or limit the amount of wires and logic. Small and simple components was important for the ability to scale to a large grid of nodes on a single chip. Epiphany's creator Olofsson discusses design decisions and trade-offs in more detail in [20].

3.2.2 Memory Architecture

Epiphany uses a distributed shared memory model; each node in the mesh contains a memory unit of up to 1MB, and all CPUs can directly address the memory unit of any node in the mesh. The first 12 bits of a memory address specify a node[1], and the last 20

[1]Note that the 12 bits that specify a node sets a hard limit of the amount of nodes that the 32-bit architecture can scale to, namely $2^{12} = 4096$ cores.

Table 3.1: Epiphany Address Space

Bits	31..20	19..0
Address	Mesh Node	Local

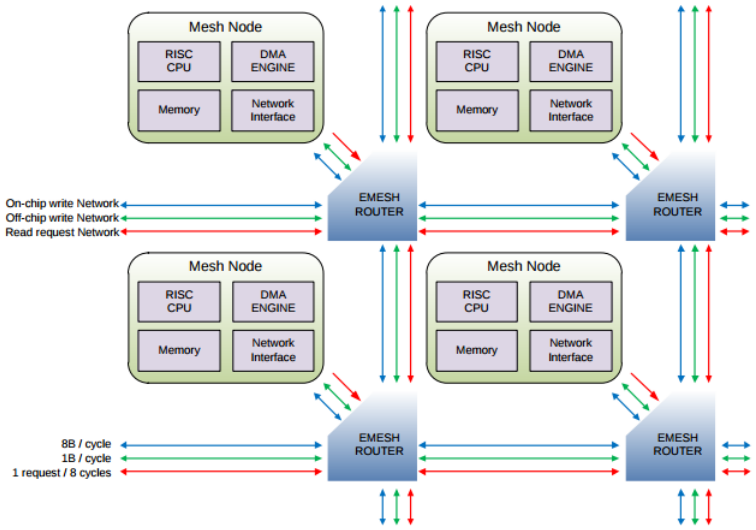


Figure 3.2: eMesh network components. (Reprinted from [2].)

bits specify the memory address local to that node.

In an effort to limit the complexity and real estate of each node, no cache is included. Instead, the memory unit in each node is divided into four memory banks that support simultaneous instruction fetching, data fetching, and intercore communication.

3.2.3 eMesh Network-On-Chip

Epiphany's Network-On-Chip (NoC), eMesh, is its most defining feature. It consists of three independent 2D mesh networks with nearest-neighbor connections. The three networks have separate internode communication responsibilities as follows: the *cMesh* performs on-chip writes, the *xMesh* performs off-chip writes, and *rMesh* handles all read requests.

Epiphany is highly optimized for moving data between nodes on the same chip by performing write requests using the cMesh. Such data transfers operate on a fire-and-forget basis. The data is written to the receiving node's memory space with no "handshake" or other transaction protocol required. A read request on data located at another local node uses the rMesh to notify the source node, which in turn performs a write request of the

data using the cMesh. Notably, this read transaction is an order of magnitude slower than the efficient write operation.

A routing module (Figure 3.2) at each node handles the stepwise flow of data through the mesh, using an address-based packet switching scheme. Message packets get routed by a deterministic path through the mesh network: first to the correct row, then to the correct column. This process, along with separate read and write networks, ensures no deadlocks in the eMesh. Routing has a sincle cycle latency per node. The networks also provide a multicast option.

The eMesh does not wrap around the edges to form a torus of nodes. Instead, it includes I/O links at each edge of its 2D mesh. This is designed to connect several Epiphany chips together to create a larger 2D grid of nodes, or to interface with other chips such as FPGAs.

3.2.4 eCore CPU

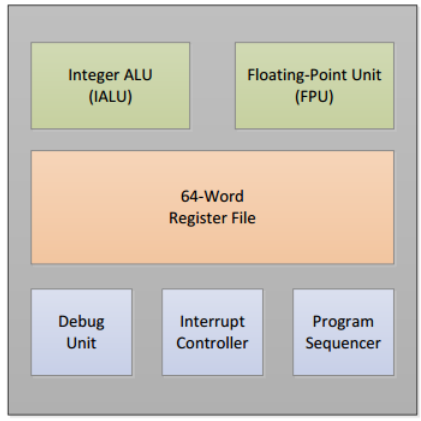


Figure 3.3: eCore processor core components. (Reprinted from [2].)

Epiphany's eCore is an in-order, dual-issue processor core that includes a floating-point unit (FPU), an interger arithmetic logic unit (ALU), and a 64-word register file (Figure 3.3). The register file may be read and written by the ALU, FPU, and a load/store instruction simultaneously. eCore CPU supports a bare-bones 32-bit instruction set architecture (ISA) and a variable length 8-stage instruction pipeline.

3.3 The Parallella Mini Computer

In 2013, Adapteva made the 16-core Epiphany-III processor widely available as part of a small single board computing eco-system called Parallella. The project was crowd-funded

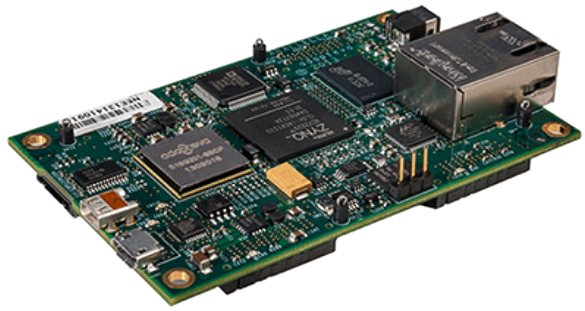


Figure 3.4: The Parallella single board computer.

through Kickstarter with the promise of making parallel, high performance computing affordable, open-source, and easy to use [1]. By 2014, Epiphany based Parallella computers had been delivered to over 200 universities around the world [20].

Parallella is used in the project of this paper for writing and running software on the Epiphany processor. The kit is supported by a software development kit (SDK) [3], and Epiphany is programmable in C/C++.

3.3.1 Hardware

Parallella uses a Zynq-7010 ARM System-On-Chip (SoC) that runs Linux, and the Epiphany processor functions as a co-processor. The board also includes off-chip RAM, Eternet, HDMI, USB, and Micro-SD storage.

The ARM CPU is typically responsible for loading software to run on Epiphany, circumventing the need for peripheral development tools.

Chapter 4

Cellular Programming with Tree Structures

This chapter describes the framework we propose for programing our parallel machine. The framework is applied to two different methods of giving input and reading output. Results are gathered in simulation, and they are shown at the end of this chapter for three different test problems. Chapter 6 further discusses these results.

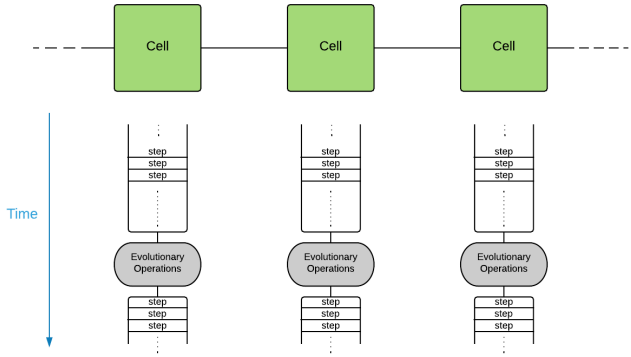


Figure 4.1: Evolutionary operations may take place locally within each cell.

4.1 Introduction and Motivation

The framework we propose for structuring a program on a parallel processor with local communication network is a combination of cellular programming and genetic programming. It mirrors the evolutionary process of cellular programming, suitable for the topology of the interconnections of the processor array. But the genetic code for cell behavior follows the genetic programming technique, suitable for a cell with full processor capabilities.

We will refer to the method as ”cellular programming with tree structures”, or CPTS for short. A cell’s behavior is defined by a tree structure of mathematical operations where each internal node is an operand and each leaf node is an input value.

CPTS is an experiment in applying concepts of CA computation to a locally connected multi-core processor and, at the same time, take advantage of the multi-core processor’s extended computational capabilities. Cells in a traditional CA has a 1-bit state value, and it’s behavior is implemented with a simple LUT. When each cell is instead a processor core, we can move from bit behavior to mathematical operations on floating point numbers, i.e. from discrete state values to continuous state values.

Cellular programming was chosen as the evolutionary framework for generating structure in our multi-core processor program because its evolutionary operations are local to each processor, or cell. For cellular programming on a LUT-based CA, these evolutionary operations must be executed on one or more capable processors external to the CA. For a multi-core processor, however, cells have the computational ability to perform evolutionary operators locally, in parallel. This is illustrated in Figure 4.1 which shows a cell array in one dimension at the top, and indicates each cell’s behavior over time. Each cell will execute its own evolutionary operations in between sets of time steps.

We use this framework on three simple problems in order to demonstrate its ability to generate a useful structure over the cell array. In particular, we look for a proliferation

of useful genes over several generations. We show results from three different problems, using two different input configurations. The three problems include:

- propagating an input stream through the machine unchanged
- propagating a modified input stream through the machine
- performing a calculation on a data set

For all three test problems, we limit the size of the cell array to 16 cells in order to match the scale of the Epiphany-III processor.

4.2 CPTS Framework

Our CPTS framework consists of a genetic coding system inspired by genetic programming, explained in Section 4.2.1, and cellular programming as the evolutionary procedure, explained in Section 4.2.2. We use two different input configurations for the framework. The first configuration operates on a continuous stream of input signals, and the second configuration receives all input before any calculation is performed. These two configurations are explained in Section 4.2.3 and Section 4.2.4, respectively.

4.2.1 Genetic Coding

The genetic code of a cell in our CPTS framework specify a tree of mathematical operations; each leaf node is an operand and each internal node is an operator. Operands may be any of the cell's input signals or zero, and operators are chosen from a predetermined list.

Each node in the tree has a gene associated with it. Our one-dimensional CPTS framework builds a tree with five nodes: three 2-bit genes encode the choice of input for each leaf node, and two 2-bit genes encode the type of function to use for each internal node. Table 4.2 shows the mapping of internal node genes, and Table 4.1 shows the mapping for leaf node genes.

Figure 4.2 shows an example tree structure, and how the corresponding 10-bit genotype is decoded. Input genes *i1*, *i2*, *i3* and function genes *f1*, *f2* are ordered from left to right in the genotype as well as in the mathematical tree structure, as shown. By consulting the two tables of gene mapping for inputs and functions, we arrive at the tree structure in the illustration.

The topology of the tree structure is predetermined, and all functions have exactly two inputs. Both inputs to a function is a floating point number, and all functions produce a floating point number. Note that an important property of the genetic goding is that any combination of bits in the genotype will allow the program to run.

Table 4.1: Set of inputs for decoding genes of leaf nodes.

code	input choice
00	center input (self)
01	left input
10	right input
11	zero (0.0)

Table 4.2: Set of functions for decoding genes of internal nodes.

code	name	function
00	add	$f(x, y) = x + y$
01	subtract	$f(x, y) = x - y$
10	difference	$f(x, y) = \|x - y\|$
11	maximum	$f(x, y) = max(x, y)$

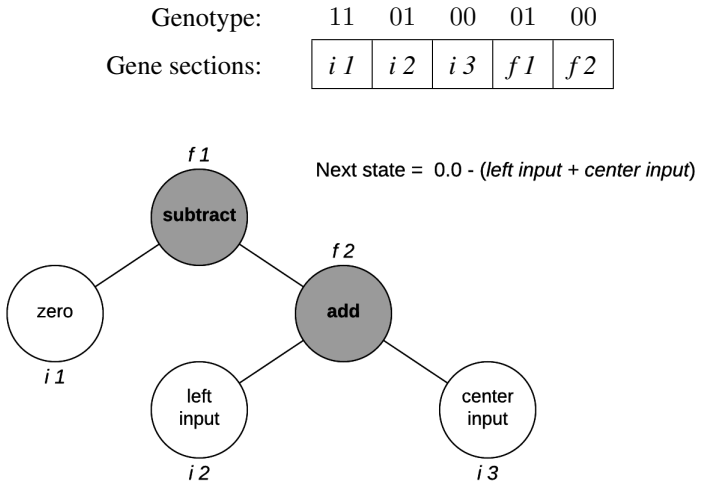


Figure 4.2: Example tree structure of a cell's mathematical behavior, decoded from a genestring.

Table 4.3: List of methods for producing the next generation of genes, based on the number of better-performing neighbors.

number of higher-fitness neighbors	new genotype	check for mutation
none	no change	yes
one	adopt the better genestring	yes
two or more	sexual recombination of two randomly picked parents	yes

4.2.2 Evolutionary Operations and Fitness

The evolutionary process of our CPTS framework mirrors Sipper's algorithm for cellular programming from Section 2.4.3. For one type of input configuration, however, we modify the point in the algorithm at which fitness is calculated. Each problem requires a custom fitness function. This is covered in the sections below.

With each new generation the population of genotypes, in our case the set of cell behaviors, is modified. Table 4.3 lists the methods by which a cell modifies its own genotype. The method depends on the number of neighbors that have a higher fitness score. If a cell has the highest fitness score in its neighborhood, it keeps its genotype for next generation. If a cell has one higher-fitness neighbor, it copies that cell's genotype. And if a cell has two or more neighbors with higher fitness score, it generates a new genotype based on sexual recombination of two of those neighbors.

After the new genotype is chosen, regardless of which method is used, it has a small chance of mutation. If a mutation occurs, it means that one or more bits in the genotype are flipped.

The evolutionary process of a cell is local to the cell, and so is the information available to it for the purpose of determining fitness. This means that the fitness function has access to input values of its cell's two neighbors only (in one dimension), in addition to the cell's own state. We do, however, allow the fitness function to store a limited history of previous input values.

4.2.3 Continuous Input Configuration

The first of two input configurations we use, we refer to as *continuous input configuration*. In this setup, the one-dimensional parallel processor array receives a continuous stream of input signals at one end, and it produces a continuous output signal at the other end. Figure 4.3 illustrates this dynamic. A data stream enters the machine from the left, propagates through the set of cells with each time step, and exits at the right side.

Note that in this configuration, the processor array does not wrap around to form a circle.

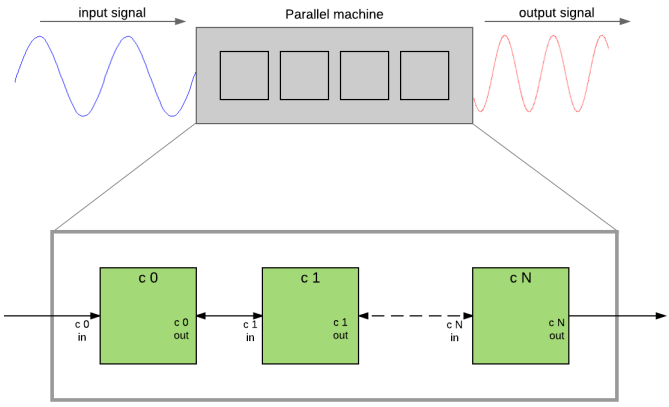


Figure 4.3: Flow of data for the continuous input configuration.

Instead, the left-most processor receives the external input signal. We have chosen to give the right-most processor a static input of 0.0 as its right-hand neighbor.

In Sipper's algorithm, fitness is calculated once for each generation, after the CA has completed its execution steps. Each cell's final state is compared with the correct result. In contrast, with a continuous input configuration, cells are not defined by their final state. Each cell performs some kind of function on an ongoing stream of input signals, and must be evaluated based on its performance at each stage of the input stream. Therefore, each cell's fitness score is modified at each time step of the machine. Algorithm 2 shows, in simplified pseudo-code, how this process proceeds and at what point fitness is measured.

Algorithm 2 Simplified pseudo-code for CPTS with *continuous* input configuration.

```
initialize cell array with random genotypes
while not done do
    initialize continuous input data stream
    intialize cell array with random initial states
    for N time steps do
        for each cell do in parallel
            perform one time step
            compute fitness based on next state of the cell
        end parallel for
    end for
    for each cell do in parallel
        perform evolutionary operations (crossover, mutation)
    end parallel for
end while
```

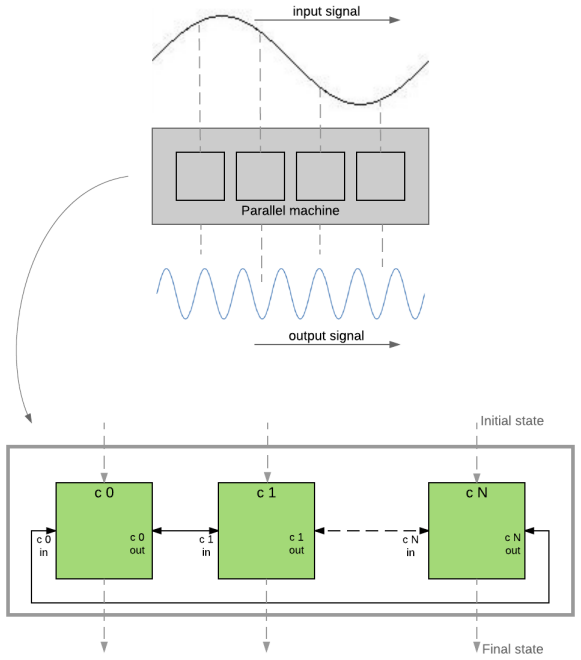


Figure 4.4: Flow of data for the single-phase input configuration.

4.2.4 Single-phase Input Configuration

We refer to our second input configuration as *single-phase input configuration*. This matches Sipper's original cellular programming work more closely with respect to how input and output is handled and how fitness is calculated (see Section 2.4.3). In this case the cell array wraps around to form a circle by connecting the right-most cell's output to the left-most cell's input.

The flow of information for this configuration is illustrated in Figure 4.4. An input signal is initially loaded onto the one-dimensional cell array, after which the machine takes no further external input. After the machine has completed all time steps, the final state of all cells is read as output.

Fitness for a cell is calculated based on the value of its final state, i.e. once per input signal. This is illustrated in Algorithm 3, for comparison with the continuous input configuration.

4.2.5 How to Read the Graphs

The behavior of the CPTS framework for the three tests below are illustrated using graphs of how dominant genes spread throughout the one-dimensional cell array. Using Figure

Algorithm 3 Simplified pseudo-code for CPTS with *single-phase* input configuration.

```
initialize cell array with random genotypes
while not done do
    initialize state of all cells according to input data
    for N time steps do
        for each cell do in parallel
            perform one time step
        end parallel for
    end for
    for each cell do in parallel
        compute fitness based on final state of the cell
    end parallel for
    for each cell do in parallel
        perform evolutionary operations (crossover, mutation)
    end parallel for
end while
```

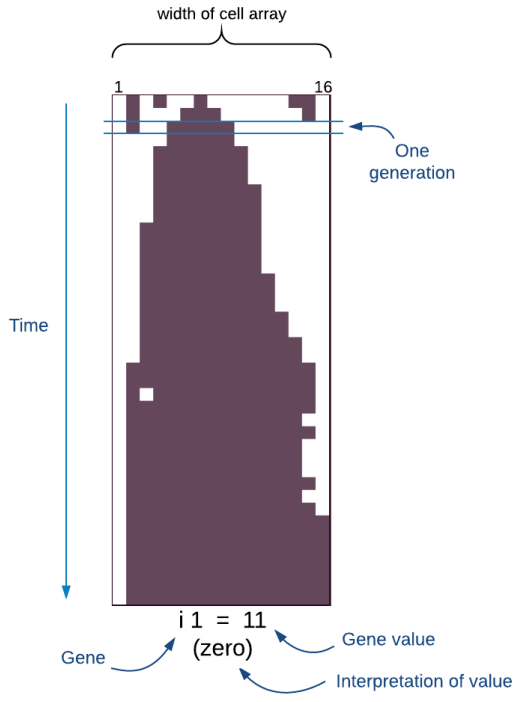


Figure 4.5: Interpretation of graphs for dominant genes.

4.5, we quickly explain how to read these graphs.

The graph is a two-dimensional array of squares that illustrate the presence of a particular gene over several generations. Each row of squares corresponds the set of rules for one generation, with the first generation at the top of the graph. Each column corresponds to the rule for one cell over many generations.

In our graphs, the processor array is 16 cells wide: cell number 1 to the left and cell number 16 to the right, as indicated above the graph. If a square is colored, it means that the particular gene is present in the corresponding cell for that generation.

The gene whose presence the graph shows is given below the graph. In the example graph, *i 1* $= 11$ means that if the gene for *i 1* has a value of "11" for a rule in the cell array, the corresponding square is colored. The semantic meaning of that gene is "zero" (as input), also indicated below the graph.

By reading the graph going down, we can see how the presence of a particular gene develops over time.

4.3 Continuous Input: Unchanged Data Stream

Our first setup that demonstrates the evolutionary behavior of the CPTS framework is a problem of passing a data stream through the machine unchanged, using a continuous input configuration. The framework arrives at a stable solution for this problem relatively quickly. In the example we show, the machine shows correct behavior after 35 generations.

The input data stream consists of data points on a sine curve. With a cell array that is 16 cells wide, we compare input data at timestep X with output data at timestep $X + 16$. If the output data matches the input data, the machine behaves correctly.

Each cell's responsibility is to propagate data towards the right. The fitness function is evaluated after each time step, and is defined as follows:

Fitness function: Unchanged Data Stream

```
function FITNESS
    if new state equals previous state then
        fitness = 0.0
    else if new state equals left input then
        fitness = 1.0
    else
        fitness = 0.0
    end if
end function
```

The fitness function first checks to see if the cell's state is modified for the next time step. This is done in order to not reward static values giving high fitness score futher down

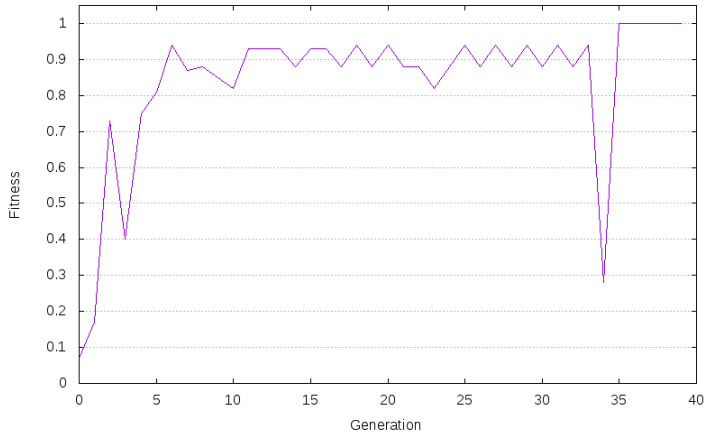


Figure 4.6: Average cell fitness for the ”unchanged data stream” problem.

the line of cells. (In particular, a static state of 0.0 for several consecutive cells would otherwise be rewarded with high fitness.) As long as state is modified, the fitness function gives a score of 1.0 if the cell's new state is set to the same value as its left input. Note that for this fitness function, there are several possible genotypes that will produce the desired output.

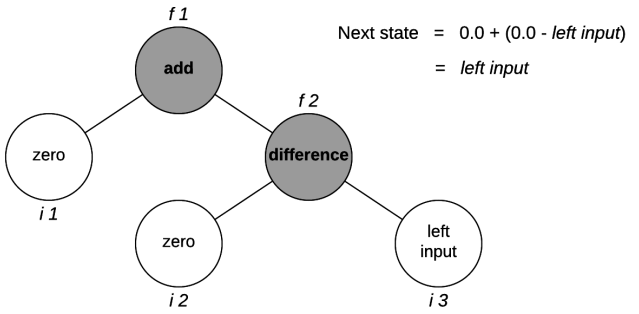
The final fitness for each cell, after all time steps are completed, is read as an average of its fitness scores for all time steps. The average final fitness score for all cells in the machine is plotted from generation 1 to generation 40 in Figure 4.6. Note that this is not an evaluation of the output result of the machine, but rather a measure of average cell performance. The figure shows that, at generation 35, all cells report a perfect fitness. For this particular problem, that also means that the machine collectively produces the desired output: an unchanged data stream.

One genotype spreads over most of the cells for the first 40 generations. It provides a fitness score of 1.0. Figure 4.8 shows its genotype, the decoded mathematical tree structure, and the equivalent mathematical function. As indicated by the figure, the genotype causes the cell to set its next state to the same value as its left input.

Figure 4.9 shows a graph of how each gene of the dominant genotype spreads across the cell array from generation 1 to generation 40. The full genotype is not represented in every cell, but the remaining cells exhibit the same behavior from a slightly different genotype.

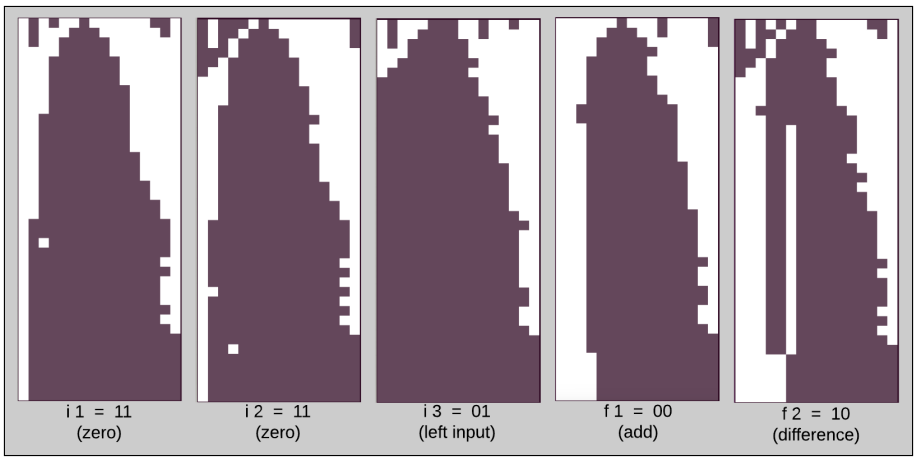
4.4 Continuous Input: Inverted Data Stream

For our second problem, the machine also receives a continuous input signal and needs to propagate this signal towards the right, but in addition we want to modify this signal. The goal of the machine is to invert the input signal such that positive input values are output



Dominant genotype:	11	11	01	00	10
	i 1	*i 2*	*i 3*	*f 1*	*f 2*

Figure 4.8: Behavioral tree of the dominant genotype for the ”unchanged data stream” problem, generations 1-40.



Dominant genotype:	11	11	01	00	10
	i 1	*i 2*	*i 3*	*f 1*	*f 2*

Figure 4.9: Development of dominant genes for the ”unchanged data stream” problem.

Table 4.4: Example of correct behavior for a cell over three time steps, for the "inverted data stream" problem.

time step	left input	cell state	right input
x	3.0	-	-
$x + 1$	-	3.0	-
$x + 2$	-	-	-3.0

as equivalent negative values, and vice versa. This problem is a bit more complicated than the first, and the solution is not stable. It illustrates important aspects of the framework that we discuss further in Chapter 6.

As for the previous problem, the input data stream consists of data points on a sine curve. With a cell array that is 16 cells wide, we compare input data at timestep X with output data at timestep $X + 16$. If the output data is a reflection of the input data, with respect to the x-axis, the machine behaves correctly.

On a local level, a cell wants to invert its left input signal, but only if its right neighbor does not also invert the signal. The cell can check for this behavior in its right neighbor. Table 4.4 shows this in a simple 3-step progression, in a situation where the center cell correctly does *not* invert the signal. This behavior gives a fitness score of 1.0 in the fitness function below.

Fitness function: Inverted Data Stream

```
function FITNESS
    if new state equals previous state then
        fitness = 0.0
    else if right input equals inverted left input of two steps ago then
        fitness = 1.0
    else if new state equals inverted previous state then
        fitness = 0.5
    else if new state equals previous state then
        fitness = 0.5
    else
        fitness = 0.0
    end if
end function
```

The fitness function makes use of a short history of input values from the left, in addition to input value from the right. If the signal is inverted only once for the center cell and the right neighbor combined, the cell gets a perfect fitness score. Otherwise, if the cell itself either inverts the signal or passes on the signal unchanged, it receives half fitness score.

The total fitness score of one generation of cells is calculated in a similar manner as for the

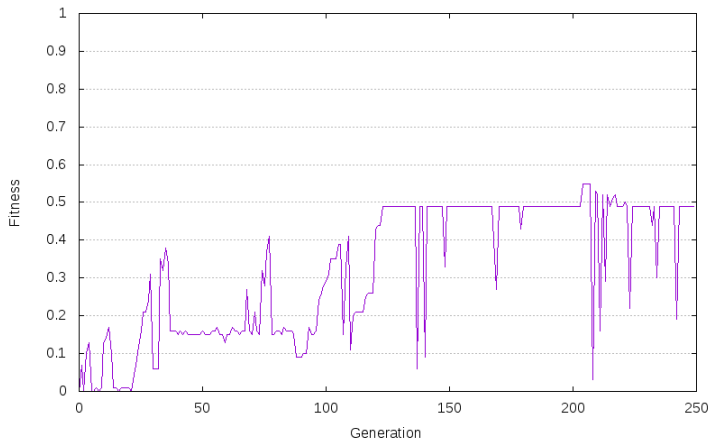


Figure 4.10: Average cell fitness for the ”inverted data stream” problem.

previous problem; it is an average of the individual cell performances. A graph of fitness from generation 1 to generation 250 is shown in Figure 4.6.

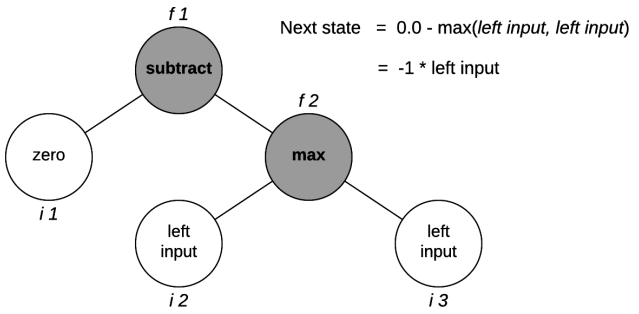
As we can see, average fitness tends to stall at a value of 0.5. At this point, cells generally behave in a uniform manner, either inverting the signal at each cell or passing on the signal unchanged. With an even number of cells, the result is the same: an unchanged signal is output by the machine. Average fitness reaches 0.5 at generation 126.

One well-performing genotype spreads in the array over the generations leading up to generation 126. Figure 4.12 illustrates the behavior of this genotype: the left input signal is inverted. Figure 4.13 shows the presence of each of its genes from generation 90 to generation 130. As we can see, four of the genes have already been established as useful genes, and the evolutionary process develops the last useful gene, *i 3*, over this period.

The most critical generations in this test, however, take place shortly after generation 200. As we can see from Figure 4.10, a short section of generations have over 0.5 in average fitness score here. We know that a small number of cells get a perfect fitness score by collaborating with their neighbor to invert the signal only once over two cells.

Figure 4.14 shows what happens in this critical section, from generation 200 to generation 220. The presence of a cell that does not invert the signal, but passes it on unchanged, gives two cells a fitness score of 1.0. This also causes the machine to behave correctly: the input signal is inverted in the output at the other end of the machine.

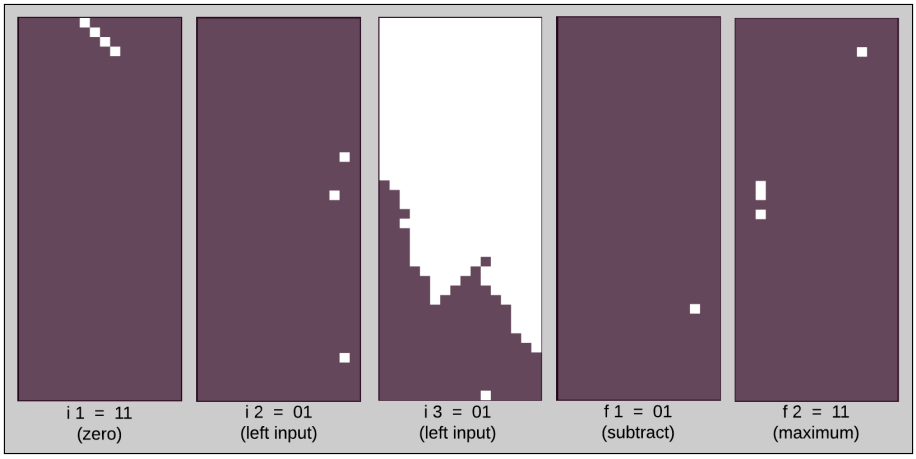
The higher fitness of the non-inverting cell causes its genes to spread, and to propagate towards the right in the cell array. Despite being responsible for correct behavior on a global level, the genotype soon goes extinct, and the machine returns to its previous uniform behavior. While the framework is able to produce a globally correct solution for this problem configuration, the solution is unstable.



Dominant genotype:

11	01	01	01	11
i 1	*i 2*	*i 3*	*f 1*	*f 2*

Figure 4.12: Behavioral tree of the dominant genotype for the ”inverted data stream” problem.



Dominant genotype:

11	01	01	01	11
i 1	*i 2*	*i 3*	*f 1*	*f 2*

Figure 4.13: Development of dominant genes for the ”inverted data stream” problem, generations 90-130.

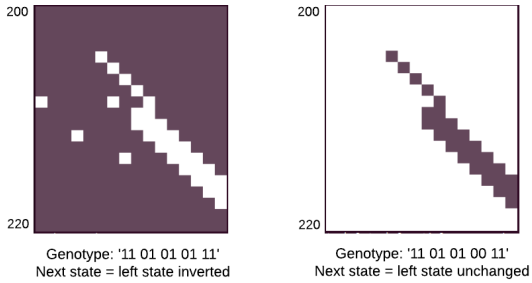


Figure 4.14: Critical section on the ”inverted data stream” problem.

4.5 Single-phase Input: Maximum Value

Our third and final problem demonstrates the framework's ability to structure a solution for a simple problem with the single-phase input configuration. Input consists of a set of randomized, positive floating point numbers, and the cell array will agree on the largest number in the array. After the machine has completed all time steps, correct behavior would result in all cells holding the same state, with a value equal to the largest input. The fitness function is as follows:

Fitness function: Maximum Value

```
function FITNESS(max input)
    if new state > max input then
        fitness = 0.0
    else
        fitness = 1.0 − new state / max input
    end if
end function
```

Each generation is evaluated on several set of randomized input data, and a fitness score is given to each cell once per input set. The average fitness score of a cell for all input sets are used for the evolutionary process. The combined fitness score for a single generation is the average score of all its cells.

The framework arrives at a stable, globally correct solution at generation 30. A graph of fitness for the first 40 generations is shown in Figure 4.15. One genotype with correct behavior spreads quickly throughout the cell array. This dominant genotype is illustrated in Figure 4.17. Its functionality is to compare the center cell and its left neighbor, and choose the largest value.

Figure 4.18 plots the growth of the set of genes of the dominant genotype for the first 40 generations. The growth of *i2, i3, f1*, and *f2* largely overlap, indicating that these genes combine to produce correct behavior, but do not necessarily contribute towards a good

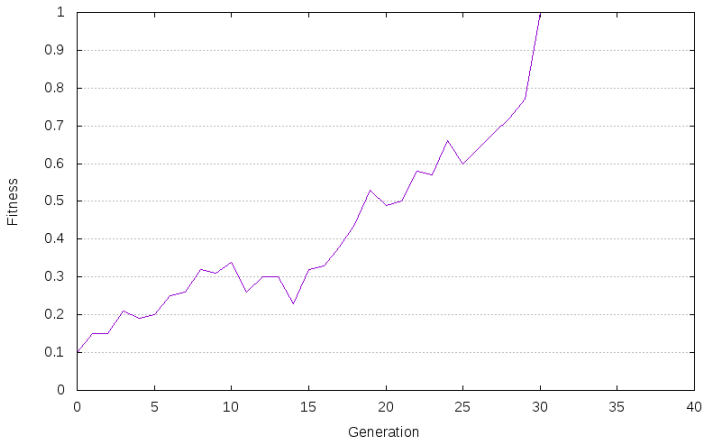


Figure 4.15: Average cell fitness for the ”maximum value” problem.

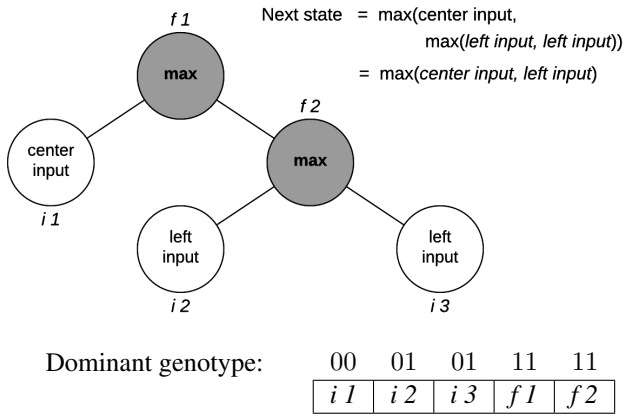
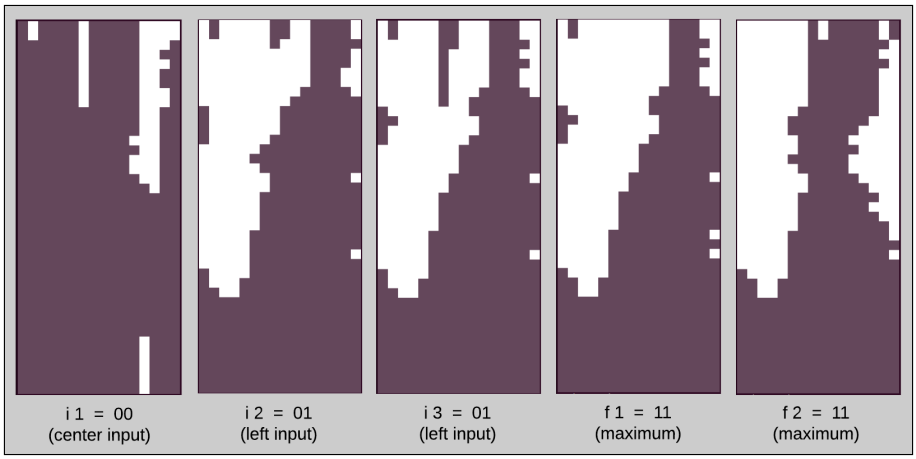


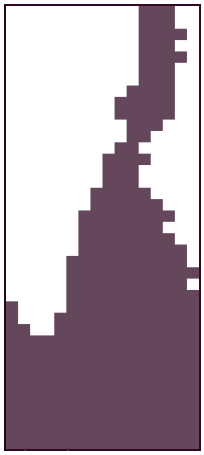
Figure 4.17: Behavioral tree of the dominant genotype for the ”maximum value” problem.



Dominant genotype:	00	01	01	11	11
	i 1	*i 2*	*i 3*	*f 1*	*f 2*

Figure 4.18: Development of dominant genes for the ”maximum value” problem, generations 1-40.

solution by themselves. A graph of their combined growth is included with Figure 4.19 for comparison.



Combination of genes:	*	01	01	11	11
	i 1	*i 2*	*i 3*	*f 1*	*f 2*

Figure 4.19: Combined growth of four important genes, generations 1-40.

Chapter 5

Implementation

This chapter gives an overview of the implementation created for the Epiphany processor. It is an implementation of the maximum value problem from Chapter 4.

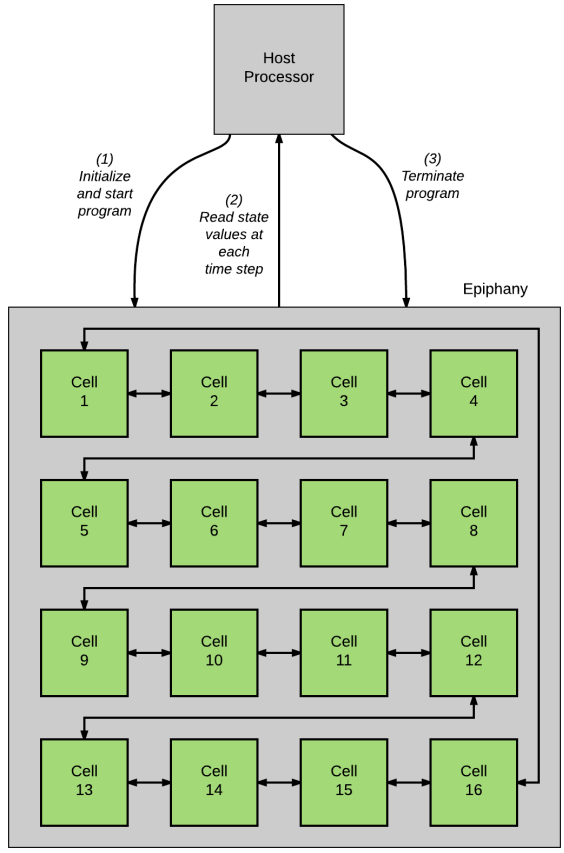


Figure 5.1: Flow of information for the implementation on Epiphany and Parallella.

5.1 Overview

The program developed by the CPTS framework for the maximum value problem is implemented on the Epiphany processor. It is a simple demonstration of how the program runs in hardware. The evolutionary procedure of the CPTS framework is not included. The implementation is designed to run on the Parallella single-board computer that hosts a 16-core Epiphany processor. Parallella facilitates development and execution of programs on Epiphany.

The implementation includes two programs written in C. The first program is run by the host processor, the ARM SoC. The second program is executed by Epiphany, and contains the parallel structure created by the CPTS framework. The flow of information between the different units in our implementation is illustrated in Figure 5.1.

5.2 Host program

The host processor's primary responsibilities are to start the parallel program on Epiphany, read its results, and eventually terminate the program. Much like a CPU typically offloads work to a GPU by giving it a function or a small program to run, the host processor on Parallella offloads the parallel program to Epiphany. It performs necessary initializations in order for the Epiphany to execute its program, such as allocating memory space, and it points Epiphany to the designated parallel program.

While the parallel program runs, the host processor reads the state values of the cells and prints them to the console for each time step. The host program also decides when to terminate the parallel program, although this is not a requirement. The parallel program may also be set to terminate on its own, according to some condition.

Procedure for the primary responsibilities of the host program on Parallella.

```
determine number of iterations (N)
initialize the parallel processor
allocate shared memory for storing results
load parallel program onto the parallel processor
for N iterations do
    read state of processor array from shared memory
    print state to console
end for
terminate the parallel program
```

5.3 Maximum Value Problem on Epiphany

The parallel program is run on Epiphany, and solves the maximum value problem described in Section 4.5. The tasks performed by the parallel program is listed in the pseudocode below. It is run in the same manner by each cell, with the exeption of how it calculates the memory address of its two neighbors.

Each cell gets its input values by reading the state of its neighboring cells. Each cell's value is located in its own local, physical memory, but the address space is shared among all cells as well as the host processor. A cell can therefore directly address the memory used by its neighboring nodes. And when it does so, the compiler asks the source node to send the required data, in this case the state value, by the eMesh to the requesting node.

The fitness function mirrors the uniform set of rules arrived at in the stable state of the maximum value problem. It chooses as its next state the highest of its own state and its left input.

The structure of how cells are connected for the one-dimensional cell array is illustrated in Figure 5.1 above. It is a simple left-to-right, top-to-bottom system, with the last cell

Procedure for the maximum value problem on Epiphany.

```
initialize position in the cell array
initialize variables for shared memory
state = random floating point number
while not terminated do
    left input = read value from local (physical) memory of left neighbor
    right input = read value from local (physical) memory of right neighbor
    if left input > state then
        next state = left input
    end if
    synchronize all cells
    state = next state
end while
```

connected to the first cell. These connections lead to more than necessary movement of data through the cell network. Other methods are more efficient, and should be considered for a more carefully constructed implementation. Efficient implementation in hardware is not the focus of this project, and the Epiphany program is created for demonstration purposes only.

Chapter 6

Discussion

This chapter provides a discussion of the CPTS framework based on the results laid out in Chapter 4. It includes a section on future work needed for the framework, and a section for hardware implementation. The final section gives concluding remarks for this project.

6.1 Structuring a program with the CPTS framework

The CPTS framework was able to program the cellular structure of our homogeneous processor array for three test problems. All test problems showed growth of useful genes over several generations. Through self-organization and an evolutionary process, the framework exploited different functions to reach the desired behavior.

As in all self-organizing systems, there is no inherently defined end to the framework's execution. The self-organizing process will run until terminated. As such, non-stable solutions may exist. This is exemplified in the inverted data stream problem.

6.1.1 Unchanged Data Stream Problem

For the unchanged data stream problem, the framework arrives at a solution fairly quickly. The responsibility of each cell is clearly defined: its input data from the left side must be passed towards the right. Constructing a working fitness function is therefore relatively straight forward.

The fitness function assigns a binary score at each time step; 0.0 for incorrect behavior and 1.0 for correct behavior. A cell's final fitness is calculated as an average of the score received at each step. When a cell is occasionally correct, the average fitness score represents partial correctness. This lets the evolutionary process identify useful genes without yet arriving at a fully correct solution.

The solution reached by the framework is a stable solution that solves this problem on a global scale.

6.1.2 Inverted Data Stream Problem

For the inverted data stream problem, the machine must modify its input signal. This problem illustrates the challenge in developing correct behavior on a global scale with a framework with local communication only. The machine manages to structure a correct solution, but that solution is not stable.

As stated, the goal of this work is to explore a framework with fitness functions that have access to local information only. The fitness of a cell must be judged according to the values available to the cell from its left and right inputs. And a lack of global information prevents the framework from consistently producing the desired global behavior for this problem.

When the structure reaches a uniform behavior in which all cells either invert the signal or propagate the signal unchanged, the structure is largely stable. This is because the fitness function gives partial credit for these behaviors. And the fitness function produces a value of 1.0 only if the cell cooperates with its right neighbor to invert the signal only once combined. With a fitness of 1.0 for cooperation, that particular behavior tends to get copied towards the right, and once again cause a double invertion or no invertion. The

cooperating behavior eventually disappears. The structure then returns to its semi-stable state.

The improved behavior, when a section of two consecutive cells invert the signal only once, is an example of differentiation in gene selections. The non-uniform behavior performs better, and is preferred fitness-wise, compared with the uniform behavior. The gene differentiation propagates to the right and fades because the fitness function does not maintain the desired structure over a larger section of cells.

6.1.3 Maximum Value Problem

The maximum value problem, using a single-phase input configuration, is set up similarly to cellular programming on CAs. Once cell states are set, the closed loop of cells runs until terminated, and the final state of cells is evaluated based on a pre-determined answer. Cells must communicate with each other in order to reach a consensus. The flow of information between the cells is less restrictive since input and output are not time-constrained nor directionally dependant. Nevertheless, the solution to the problem propagates in one direction only.

This problem exemplifies a class of problems that has a local behavior that directly contributes to a global, coordinated effort. Equal fitness evaluation encourages even behavior among the different cells, and the even behavior results in a globally acceptable solution. As we discuss later in this chapter, achieving diverse cell behavior that collectively cooperate to achieve a result is difficult, and is reliant on excellent fitness evaluation.

6.2 Challenges of the CPTS Framework

Constructing a framework for self-organizing a program on a parallel machine is difficult. We can identify two major challenges that must be addressed in this type of framework. The two challenges relate to a lack of global communication and the effectiveness of a uniform fitness function for all cells. Nevertheless, these challenges must be overcome, not circumvented, in order to make progress towards the goals of this research topic.

6.2.1 Lack of Global Information

Cellular programming evaluates each cell individually, using information available to that cell only. The limited scope of the fitness function's knowledge has implications for the type of problems the framework is able to solve. This is illustrated by the inverted input data problem in Chapter 4. While a cell can evaluate the collective correctness of its immediate neighborhood, it has no knowledge of the size of the cell array. Therefore, it cannot ensure global correctness.

This issue could quite easily be solved by introducing a small amount of global information to the fitness functions. But while global information is possible to do, it is not relevant

for the goal of this research. We aim to solve problems at the global scale using local interactions exclusively. Cellular programming seems to be limited to a class of problems where each cell's contribution to the solution can be evaluated locally.

Note that this restriction does not exclude interesting cooperative behavior or differentiation on gene selections, however. In the cellular programming research with CAs (see Section 2.4.4), cells communicated and agreed on behavior at a global scale by way of local communications.

6.2.2 Uniform Fitness Function with Non-uniform Rules

While our cellular array may include a variety of differently behaving rules, the fitness function is uniform across all cells. That causes cells tend to strive for uniform behavior, and brings into question the usefulness of a non-uniform rule environment. We know from previous work with CAs that cellular programming with uniform fitness functions may produce varying cell behaviors that lead to successful collective cooperation. Achieving this dynamic requires more work in structuring fitness functions and problem representations in our framework.

Uniform fitness functions is a necessity for this research. If we use varying fitness functions for our cell array, we are essentially programming the machine manually. The principle of self-organization would not be present, and the result would not be relevant for our goals.

6.3 Future Work

Introducing the CPTS framework and demonstrating its features on the test problems of Chapter 4 is only a minor step in the direction of our overarching research goals. The framework needs a significant amount of further work for it to be useful for solving interesting problems, and the challenges ahead are difficult to solve. The most important work going forward with framework involves problem representation and fitness functions. Further exploration into these topics was out of reach for this project.

6.3.1 Problem Representation

A critical attribute of frameworks of this type is evolutionary adaptability: its ability to self-organize through a variation and selection process. While a cellular model forms a good basis on which build an evolutionary adaptable framework, more research is required on how to effectively represent behaviors and functionality.

The framework must be flexible enough to support a large variety of behaviors, and it must facilitate gradual changes in behavior. The choices for cell neighborhood, tree topology, and gene mapping play a role these aspects. Whether non-static tree topologies, to name

an example, may lead to greater flexibility in the framework is one of many possible paths to explore.

6.3.2 Fitness Functions

Constructing a fitness function for local behavior, that will ultimately result in a desired global behavior, is a difficult problem. A better understanding of what makes for a successful fitness function is critical to the usefulness of non-programmable, evolvable frameworks.

While the issue of problem representation relates to the framework in general, the fitness function is specific to the problem we want to solve. Select problems may be able to exploit more easily available fitness functions, but in order to apply the concepts investigated in this project to a broad set of problems, we require more research in how to evaluate local cells effectively.

6.4 Implementation

Implementation of the program structured by the CPTS framework is primarily for demonstration purposes. Currently, obstacles like effective problem representation and fitness evaluation limit the CPTS framework's usefulness. As such, further work should be concentrated primarily on making progress in those areas. When promising results are found, a complete implementation that incorporates the principles shown in this project can result in a working prototype in hardware.

6.5 Concluding Remarks

In this project we introduced a cellular programming-based framework designed for homogeneous parallel processors. We demonstrated its ability to spread useful genes and develop a global behavior based on local interactions and evaluations. The results show that the self-organizing process of the framework can program such a processor array. The software organized by the framework was executed on the Epiphany parallel processor.

Three different problem configurations were solved by the framework, but not all of the solutions were stable. Global information is unavailable to the cells, correct local behavior may not necessarily guarantee correct global behavior. The problems also illustrate challenges involving problem representation in the framework and contruction of effective fitness functions at the local level. It is difficult to conclude the potential of this framework without further progress with respect to these challenges.

Bibliography

[1] Adapteva. Parallella kickstarter project. https://www.kickstarter.com/projects/adapteva/parallella-a-supercomputer-for-everyone/, 2012.

[2] Adapteva. Epiphany architecture reference, 2013. Available at http://www.adapteva.com/docs/epiphany_arch_ref.pdf.

[3] Adapteva. Epiphany sdk reference, 2013. Available at http://adapteva.com/docs/epiphany_sdk_ref.pdf.

[4] ARM. big.little technology moves towards fully heterogeneous global task scheduling, 2013. Available at https://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf.

[5] K. Asanovi, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] M. Conrad. The universal turing machine (2nd ed.). chapter The Price of Programmability, pages 261–281. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[7] P. A. Corning. The re-emergence of emergence: A venerable concept in search of a theory. *Complexity*, 7(6):18–30, 2002.

[8] J. P. Crutchfield and M. Mitchell. The evolution of emergent computation. *Proceedings of the National Academy of Sciences*, 92(23):10742–10746, 1995.

[9] T. De Wolf and T. Holvoet. Engineering self-organising systems. chapter Emergence Versus Self-organisation: Different Concepts but Promising when Combined, pages 1–15. Springer-Verlag, Berlin, Heidelberg, 2005.

[10] R. Doursat, H. Sayama, and O. Michel. A review of morphogenetic engineering. *Natural Computing*, 12(4):517–535, 2013.

[11] M. J. Flynn. Parallel processors were the future ... and may yet be. *Computer*, 29(undefined):152,151, 1996.

[12] F. Heylighen. The science of self-organization and adaptativity. *The Encyclopedia of Life Support Systems*, pages 1–26, 2001.

[13] W. Hordijk, J. P. Crutchfield, and M. Mitchell. Mechanisms of emergent computation in cellular automata. In *International Conference on Parallel Problem Solving from Nature*, pages 613–622. Springer, 1998.

[14] A. Khuong, G. Theraulaz, C. Jost, A. Perna, and J. Gautrais. A computational model of ant nest morphogenesis. In *Proceedings of the Eleventh European Conference on the Synthesis and Simulation of Living Systems, Advances in Artificial Life, ECAL2011*, pages 404–411, 2011.

[15] C. G. Langton. Computation at the Edge of Chaos: Phase Transitions and Emergent Computation. *Physica D*, 42:12–37, 1990.

[16] E. Manley and T. Cheng. Understanding road congestion as an emergent property of traffic networks. In *Proc of 14th World Multi-conference on Systemics, Cybernetics and Informatics*, pages 25–34, 2010.

[17] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5):33–35, Sept 2006.

[18] M. Moses, T. Flanagan, K. Letendre, and M. Fricke. Ant colonies as a model of human computation. In *Handbook of human computation*, pages 25–37. Springer, 2013.

[19] A. Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip, 2016. Available at `http://www.parallella.org/docs/e5_1024core_soc.pdf`.

[20] A. Olofsson, T. Nordström, and Z. Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726. IEEE, 2014.

[21] D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA, 1986.

[22] M. Sipper. The Emergence of Cellular Computing. *Computer*, 32(7):18–26, 1999.

[23] M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[24] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, and A. Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *Evolutionary Computation, IEEE Transactions on*, 1(1):83–97, 1997.

[25] C. Taylor and D. Jefferson. Artificial life as a tool for biological inquiry. *Artificial Life*, 1(1_2):1–13, 1993.

[26] G. Tufte. From Evo to EvoDevo: Mapping and Adaptation in Artificial Development. *Evolutionary Computation*, (October):219–238, 2009.

[27] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002.