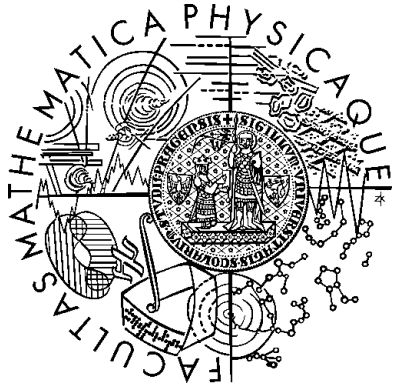
Charles University in Prague
Faculty of Mathematics and Physics

**DOCTORAL THESIS**




Lukáš Petrů

# Universality in Amorphous Computing

Institute of Computer Science
Academy of Sciences of the Czech Republic

Supervisor: Prof. RNDr. Jiří Wiedermann, DrSc.
Branch: i1 – Theoretical Computer Science

I would like to thank my supervisor, Prof. RNDr. Jiří Wiedermann, DrSc., for bringing the field of amorphous computing into my attention and for constant support and invaluable advice that he provided to me throughout my doctoral study.

Title: Universality in Amorphous Computing

Author: Lukáš Petrů

Department: Institute of Computer Science, AS CR

Supervisor: Prof. RNDr. Jiří Wiedermann, DrSc.

Supervisor's e-mail address: wieder@cs.cas.cz

Abstract: Amorphous computer is a theoretical computing model consisting of randomly located tiny devices (called nodes) in some target area. The nodes of an amorphous computer can communicate using short-range radio. The communication radius is small compared to the size of the target area. The nodes are all identical, initially have no identifiers, work asynchronously and there is no standard communication protocol. An amorphous computer must work for any number of nodes under reasonable statistical assumptions concerning the spatial distribution of nodes. Moreover, the computation should use very limited amount of memory on each node.

For the just described concept of amorphous computer we investigate the question whether a universal computation is possible at all in a corresponding theoretical model. To answer this question, several subsequent steps are performed. In the first step, we design a formal minimalist model of a node and of the amorphous computer as a whole. In the second step, we develop communication protocol for the amorphous computer. In the last step, we show the universality by simulating a computation of a universal machine. The size of the amorphous computer will depend on the space complexity of the simulated machine. All the previously mentioned steps are described in detail in this work. Actually we present three models of amorphous computers, each differing in some aspect. While the first model deals with densely placed nodes, the second one deals with nodes sparsely placed in the target area. The third model deals with mobile nodes. We show the universality for all three models. To the best of our knowledge, this is the first result of this kind for minimalistic computational models of amorphous computers.

Keywords: amorphous computing, computational complexity, universality

# Contents

# Chapter 1

# Introduction

Nowadays, the progress in fabrication of electrical and mechanical components has reached the point where it is possible to build a device containing sensing, actuating and information processing components within a volume of several cubic millimetres. Such devices can be distributed in our everyday environments to help with various tasks [5], [6]. For instance, by sensing weather-related data (temperature, air pressure, humidity and other) at several locations, they help for more accurate weather forecasts. In agriculture, data from sensors can be used to achieve better yield of plants. On roads, the sensors can help with better traffic control and with locating parking places. In secured buildings, such sensors would enable people and item tracking.

Ideas of tiny devices freely floating in the air which are either controllable by humans or have their own intelligence have already appeared in sci-fi literature [34]. However, manufacturing such a device requires overcoming of several obstacles. The semiconductor industry has been able to continually decrease the size of the produced integrated circuits, and today it is possible to produce a simple microprocessor in a cubic millimetre size. Recently, the technology of microelectromechanical systems (MEMS) also enabled construction of parts requiring mechanical components—such as various sensors and actuators. So, in principle the technology is ready in these two areas. The remaining technological problems to be solved are in providing enough electrical energy to operate the devices and enabling them to fly in the air. Then, we will have the devices dreamed of in the sci-fi literature.

Several scientific projects exploring these and similar ideas have already started. They deal with designing the hardware of the devices, searching for programming paradigms, describing computational models and analysing their power. In this chapter we briefly introduce three of the projects—Smart dust, Amorphous computing and Population protocols.

The research focus in this area has mainly been on the properties that set these kinds of systems apart from other known systems—the possibility to gather data from a large number of spatial positions, aggregating the data, performing spatially-restricted computation and actuation around a place where some event occured. The idea is that the individual devices operate on data sensed at their actual positions and react locally. Apart from planning actions locally, the devices can also send data to some base station that provides large-scale control of the whole system.

In the rest of the chapter we present an overview of several projects preceding and inspiring our work. These projects will serve both as a framework giving a context to our present work as well as a reference basis towards which our results can be compared. The first project was dedicated to designing the node hardware. The second project investigated a new programming paradigm—amorphous computing—and also introduced a model which has served as the main inspiration for this work. The third project is a theoretical research with similar goals as ours but using a different underlying model.

## 1.1 Smart dust

### Description

The Smart dust project has run at the University of California, Berkeley in years 1998–2001, led by K. S. J. Pister. The project goal was to develop a self-contained miniature device that would contain an energy source, a processor, memory, an optical transmitter and various sensors or actuators. Intended use of such devices was to spread a large number of them in some area to monitor the conditions there and send the findings to a remote site using a long-range optical communication. Several academic and commercial projects have tried to advance the idea up to the point of practical application. One of the groups currently active in the area is, e.g., the Speckled Computing consortium [45] in UK.

### Results

The Smart dust project was probing the contemporary technology limitations in production of miniature self-contained sensor network devices, called motes (see [36]). A mote consists of the following components: battery, microprocessor, memory, communication system, sensors and actuators. The main interest was in building such a device as small as possible. The size constraint, when applied to battery, transforms in turn into energy constraint on

all other parts. The past advancements in microelectronics enabled shrinking of electronic devices and lowering of electronic consumption at the same time. Using the contemporary technology the Intel 8088 core could be manufactured 100× smaller and 100× more energy efficient compared to the technology by which it was originally produced. The energy efficiency can further be increased by designing custom microprocessor specifically targeted for this low-energy scenario. As part of the Smart dust project a microprocessor consuming only 12 pJ per instruction was developed (see[37]) .

Two means of mote communication were considered: communication using radio waves or optical communication. Using optical transmission, it is possible to concentrate all energy into a narrow beam that can be directed at the receiver. With the radio transmission, the energy is spread in all directions, so it is less effective. Due to the energy constraints the Smart dust project concentrated on the optical communication (see [25], [26]). The mechanical parts of the communication system can be manufactured at very small size scale using the MEMS technology.

The Smart dust project initially set out with an ultimate goal of a mote in 1 $mm^3$ volume. That particular size was not achieved. However, two test nodes of volumes 138 $mm^3$ and 63 $mm^3$, respectively, were presented (see [35]).

In 2002, after the Smart dust project finished, Kris Pister founded Dust Networks[44], a company to promote the idea to a commercial application.

Currently, there are many institutions carrying out the research in similar area, one of them being the Speckled Computing consortium[45] focusing on further miniaturization of the devices and their components. In future these devices called specks could be sprayed on everyday life objects to endow them with sensing, processing and wireless networking capabilities [12].

## 1.2 Amorphous computing

### Description

The research on amorphous computing started at MIT Computer Science and Artificial Intelligence Laboratory in 1996 by H. Abelson, T. F. Knight, Jr., G. J. Sussman, et al. The amorphous computing project concentrated on systems built from a very large number of identical components, which are randomly placed, locally interconnected and have limited computing capability (see [2]). The building components can be electrical devices, but that is not a requirement. Neural networks, multicellular organisms and chemical systems can also be viewed as amorphous systems. The goal of Amorphous

computing was to find suitable programming paradigms that could be used to control the operation of an amorphous system.

## Results

The Amorphous computing project dealt with two complementary parts. One part was hardware, the other one was software. A possible hardware could have been miniature electronic devices, similar to those used in Smart dust. One prototype computer of this type was built, so-called pushpin computer (see [18]). The devices had a shape of pushpins and they were stuck onto random positions on a conductive board. The conductive board provided the power for the nodes and provided a shared medium for communication.

Another possible hardware could be made of living cells. A cell could be programmed by modifying its genome. Cells could communicate with other cells through signalling mollecules. An organic amorphous computer could be created by producing a large number of cells with identical genome and putting them into a container filled with some liquid medium. Initial research was already done in this direction. It was shown that simple logic gates (such as AND, OR and NOT) or oscillators could be created (see [38]).

Systems of living organisms occuring in the nature, such as multicellular organisms or swarms of insects, can also be seen as instances of amorphous computing systems. In an organism all cells have the same genome and in this sense are identical. Also a swarm of insects, such as bees, consists of physically identical members (except for the queen bee). Such systems rely on a large number of elements acting and communicating locally yet the whole exhibits a common global behaviour. It is obvious that these natural systems cannot be easily reprogrammed. Yet, they can serve as a valuable source of organizing principles that could be applicable also to man-made amorphous systems.

Research on amorphous computer programming was not done directly on any real platform, but on a simulator. In the simulator, several thousand nodes are simulated, each operating as if it had a processor and memory. The simulator also handles message exchange among nodes that are in communication range.

To the programmer, the simulator presents the following computational model of an amorphous computer (quoted from the Amorphous computing manifesto [1]):

> *An amorphous computing medium is a system of irregularly placed, asynchronous, locally interacting computing elements. We can*

> *model this as a collection of "computational particles" sprinkled irregularly on a surface or mixed throughout a volume. The particles are possibly faulty, sensitive to the environment, and may effect actions. In general, the individual particles might be mobile, but the initial programming explorations described here do not address this possibility.*
>
> *Each particle has modest computing power and a modest amount of memory. The particles are not synchronized, although we assume that they compute at similar speeds, since they are all fabricated by the same process. The particles are all programmed identically, although each particle has means for storing local state and for generating random numbers. In general, the particles do not have any a priori knowledge of their positions or orientations.*
>
> *Each particle can communicate with a few nearby neighbors.*

Now we move to the description of the software part, i.e., what algorithms were proposed for controlling the behaviour of an amorphous computer.

The earliest attempts were to simulate a behaviour of growing shapes, similar to the processes observed in developmental biology. The amorphous computer starts with all but a few computing particles (we call them nodes) in a common initial state. A few nodes start in special states, one node that marks the place where the growing starts from and few others that determine the orientation of the shapes. According to a common program that is loaded into all nodes, the nodes communicate with their neighbours and start growing different regions. A region is a set of nodes that are in some special internal state. Continuing with the developmental biology analogy, a region corresponds to a tissue. Initially, the region starts with a single node in a special state; this node later adds its neighbours to its region by sending them a message. The node can control which neighbours to add, so the regions can be grown to some prespecified shape. The program for region growing has been given in a special language called GPL, invented by Coore (cf. [19]). The shape growing process is quite robust and adapts to the unknown, random distribution of the nodes.

Another work drawing inspiration from nature was that of R. Nagpal (cf. [27]). In her work, she simulated how living tissues fold to create complex spatial shapes. The amorphous computer particles are assumed to be embedded into a bendable sheet and each node has actuators that can bend the sheet around the node. According to a common program loaded into all nodes, the nodes compute where to form a bending line and nodes on this bending line start bending the sheet. After performing several bendings,

complex shapes can be formed. It was shown that it is possible to exactly reproduce a process of paper folding as is done in the japanese Origami.

If a message is sent from one node to its neighbours, and from these neighbours to their neighbours and so on, and if the message keeps a record of the number of hops made, the node receiving the message can deduce an approximate distance to the message source. This algorithm is called a *gradient*. Given three anchor nodes whose positions are known and starting three gradients from these nodes it is possible to construct a coordinate system on an amorphous computer. This has been presented by Nagpal [28].

To bring a higher layer of abstraction for amorphous computer programming, Beal proposed a programming language called Proto (see [14]). The Proto language enables to easily start a gradient at some node. The starting of a gradient and similar basic actions can be triggered by various logical conditions based on sensory input and messages from other nodes.

Using slightly more powerful nodes, Butera [18] introduced the concept of a paintable computer. In that scenario, nodes are able to run small Java programs called fragments. These fragments can freely migrate from one node to another and so hover throughout the whole amorphous computer. Realization of more complex programs requires the program to be split into several fragments and these fragments must then find each other inside the amorphous computer to perform the operation. Butera presented several applications for his paintable computer, such as audio or image storing, network connectivity for external devices and image segmentation.

Lately, there has been interest in designing a high-level programming language for an amorphous computer that would abstract-out individual nodes and instead allow the programmer to treat the computer as a space-filling medium [15], [3]. This new language extends the work on Proto language mentioned above. The language should provide some primitives, such as Gossip (epidemic communication), Random choice (used for symmetry breaking), Gradient, and other. Then, it should provide means of combining these operations, such as carrying out a sequence of operations or restricting the operations to some spatial region.

## 1.3 Population protocols

### Description

In 2003, a new idea of so-called population protocols was introduced by Angluin, Aspnes, et al. ([7]). The population protocol is a model of a network of finite-state agents that randomly communicate in pairs. The

model reflects the scenario when miniature sensors serve as agents attached to some independently moving carrier (e.g. sensors are attached to birds). As the carriers move in the space randomly, the sensors that come near to each other create a pair that can exchange information. The aim of this research is to find the computational power of this model and its variants.

## Results

First we describe in more detail the formal model of population protocols as it was given in [8]. The model consists of a set of agents, finite input and output alphabets $X$ and $Y$, a finite set of states $Q$, a function $I : X \to Q$ mapping inputs to states, and a function $O : Q \to Y$ mapping states to outputs. Each agent is in some state $q \in Q$. The operation of the protocol is specified by a transition function $\delta : Q \times Q \to Q \times Q$, which takes states of two interacting agents and assigns new states to these agents. A population configuration consists of a multiset of elements of $Q$ specifying the state of each agent. A computation is a finite or infinite sequence of population configurations. In one step of a computation two agents of an old configuration are selected, and a new configuration is produced by substituting the old states of the two agents with new states given by the transition function $\delta$. The two agents that interact in one step are selected by some external adversary. However, the adversary must obey pairwise-fair criterion. The computation is pairwise-fair if it satisfies the following property: when given two configurations $C1$, $C2$ such that it is possible to go from $C1$ to $C2$ in one step and when $C1$ is encountered infinitely many times in the computation then the transition to $C2$ will also take place infinitely many times throughout the computation. This criterion ensures that if there is a special pair of agents whose interaction is essential for the progress of the computation the adversary may delay their interaction for a finite time but not infinitely.

In this setting we cannot say when a computation has actually finished. However, for some population protocols the output of the agents in an infinite computation eventually stabilizes. The computational power of such a system can be characterized by the class of functions that are eventually computable by the system. We say that a population protocol $A$ eventually computes function $f$ if for every input $x$ and every pairwise-fair computation of $A$ starting with the corresponding input configuration $I(x)$, the output stabilizes to $f(x)$.

Allowing two output values 0 and 1 for each agent enables the population protocol to compute a predicate. The interesting question of which predicates can be eventually computed by a population protocol has been answered in [10]. It is shown that the stably computable predicates are equal to a class

of semilinear predicates. Semilinear predicate can be written as a logical formula using symbols $<$, $+$, 0, 1, and the standard logical quantifiers and connectives. For example, the population protocol can decide whether there are more agents in some state $q_0$ that in state $q_1$, or whether the number of agents in state $q_0$ is even. Clearly, the model lacks universal computational power. The main reason for this is that sequence of computation steps cannot be performed since it is not known when each step would finish.

A variant of the computational model, called conjugating automata, was introduced in [8]. Here, the pair of agents to interact on each step of computation are selected randomly instead of being selected by an adversary. This allows to compute the probability that after a given number of steps all agents have interacted with each other. It also enables to run an operation for a fixed number of steps, after which the output should be ready, and another operation can be run. So, it is possible to perform a sequence of actions. The set of agents can simulate a fixed number of counters where a counter stores a unary coded number. For example, the contents of one counter can be represented as the number of agents that are in state $q_1$. Having counters and the possibility to perform a sequence of operations it is now possible to simulate a computation of a counter machine (see [9]). However, we cannot guarantee that all operations of the counter machine finish in their alloted time steps, so there is always some probability of error in the computation. This error can be made arbitrarily small by providing longer time for each operation.

It is possible to define several variations of the basic population protocol model. Recently, another mode of computation—so called self-stabilizing population protocols (see [11])—was considered. Self-stabilizing protocol is such a protocol that achieves a correct configuration of agents' states even when starting from some incorrect configuration. The protocols work in a setting where agents do not interact with other random agents but possible interactions are given by a simple graph, e. g. by a ring. An example of such a protocol would be one that chooses single leader of all nodes. If for some reason the configuration of agents would change and there would be none or more than one leader, the self-stabilizing property of the protocol would later again recover the correct configuration with only one leader.

## 1.4 Comparison with our approach

This work was largely inspired by the amorphous computing idea. For instance, the concept of a computer that can actually be painted on any sur-

face is very enticing and provides a good research theme. Looking at what research had already been done we observe that the focus was on the characteristics that set this new computing system apart from conventional technologies. The initial experiments explored various spatial properties of an amorphous computer—how it can measure distances, how it forms a path between two spatial points or how it grows regions of a prespecified shape. However, no attention has been paid to comparing amorphous computer with conventional computers in terms of the computing power. We see this as a gap that we would like to fill within our work. We provide the proof that reasonable models of families of amorphous computers have universal computing power.

None of the three projects mentioned before presents the results that we are after. So far, the research in amorphous computing was focused mainly on analysing, how an amorphous computer could perform certain concrete applications. The question of universal computational power was neither under investigation nor answered in any, perhaps indirect, way. The research on amorphous computing employed a bottom-up approach, starting with primitive operations and extending them to higher-level actions, without a clear knowledge about the limits of what is possible.

The Smart dust project dealt mainly with the technical problem of building a tiny sensor node and constructing a network of such nodes. For our work the Smart dust project serves as an illustration that the idea of such systems is practically realisable and promising for the near future.

The Population protocols research is in its goals quite similar to our work—here a formal model is defined and its computational power is investigated. However, from our point of view the model is not in a close correspondence with the model of smart dust or amorphous computing, which we used in our work. On the one hand, the population protocols' nodes use a constant amount of memory, which prevents the nodes to have unique addresses. Consequently, the universal computation is only enabled by a simulation of a counter machine, which is very slow for any practical use. On the other hand, the communication subsystem allows exactly two nodes to mutually exchange data. That is quite a strong assumption—if there are multiple nodes, then picking exactly two to take part in the communication is a nontrivial task. In our communication model the communication is unidirectional and the sender never reliably knows if there was some receiver to receive the data. Hence, our communication model is weaker and we think it better corresponds to a possible implementation of an amorphous computer. To conclude, our approach is in the spirit of computational, or complexity theory, giving a rigorous formal treatment for algorithm design and complex-

ity analysis. As a result, our models can be embedded among the existing body of knowledge within these fields and enable their comparison with other models considered in the theory.

# Chapter 2

# Aims and contents of this work

Amorphous computing is built on the idea of setting-up a computer by gathering a large number of simple, identical, computational and communicating elements in a bounded area. Thanks to the simplicity of this scenario it seems possible to make a theoretical model of an amorphous computer and study its properties. To the best of our knowledge, so far this has not been the case and, consequently, the power of an amorphous computer was not yet analyzed from the theoretical computer science point of view. This is what we will do in our work.

Our main goal is to investigate whether the amorphous computers could be designed so that while preserving the minimalistic requirements concerning their realization they have a universal computing power, i. e., whether they are able to simulate other universal models known from the computational theory. While the universal computational power will probably not be needed in a concrete application of an amorphous computer, it serves as a proof of concept that an arbitrarily complex computation can be carried out by an amorphous computer. So, the result of universality proves that the model can also be used in any other application of amorphous computer. This goal calls for a number of design decisions related to, e. g., the memory capacity, random number generator availability and the properties of the radio transceiver.

To show the universal computing power of an amorphous computer we must go through several steps, each step building on the preceding one. The necessary steps are the following ones:

1. formally define the model of an amorphous computer,
2. develop a communication protocol for the model,
3. analyse the properties of the communication protocol,

4. using this protocol, describe a simulation of some universal machine (e.g. of a cellular automaton or a RAM) on the proposed model of an amorphous computer.

To illustrate the spectrum of problems we are facing, consider the problem of time complexity of amorphous computations. To determine the time complexity of the computation of our model we need to analyse the time complexity of the underlying communication protocol of the model. The speed of the communication protocol in turn depends on the actual positions of the nodes, which are random. To understand the effect of random node positions on communication speed we need to investigate properties of random graphs, such as their diameter, component size, and maximum neighbourhood size. Some of these properties are difficult to estimate analytically and we will have to rely on computer simulations.

When building a model of an amorphous computer, there are several design choices that we can make and that lead to different models. One choice concerning the nodes of an amorphous computer is whether they will be stationary or mobile after being deployed. Another choice is whether the nodes will be deployed densely or sparsely. In our work, we will inspect all these possibilities. We start by analysing the model of densely placed stationary nodes. Afterwards, we develop and analyse a model with sparsely placed stationary nodes. Eventually, we study a model with mobile nodes. All three models are presented in this work, each model in one chapter.

The three models differ not only in their "architecture", but also in the protocols and algorithms they use. The protocols and algorithms must be adapted because each new model weakens some property of the previous model. The first model (Model 1) enables to map data onto a regular grid of cells, where no global addresses of the cells are needed. When going from this model to the second model (Model 2), which uses sparsely placed nodes, the regular grid cannot be formed and so for distributing data among the nodes we must equip the nodes with a mechanism allowing them to generate their individual addresses. Thanks to the fact that in Model 2 the communication links between nodes are static, we can compute the probability of a communication error and the necessary number of retransmissions. Finally, in our third model (Flying amorphous computer), the links are dynamic and there is no guarantee that a message will be delivered with some fixed probability. In that case we must use a new protocol that requires explicit acknowledgements of the computation steps done by individual nodes, so that the amorphous computer does not enter an incorrect state when a message is lost.

All presented models are non-uniform. The required size of an amorphous

computer depends on the space complexity of the algorithm being run, which, in turn, depends on the input size.

Model 1 is introduced in chapter 3. This model makes use of densely placed static nodes. "Densely placed" means that the nodes cover the target area without leaving any significant holes in it. We show that on this model it is possible to form regions organized in a regular square grid. In such a case, it is possible to simulate a computation of a cellular automaton on this model. Model 1 enables a simulation of a parallel computation, whereas all other models described in subsequent chapters only enable simulation of a sequential computation.

Model 2 is introduced in chapter 4. This model makes use of sparsely placed static nodes. Sparse placement of nodes leads to a lower rate of communication collisions and also to energy saving in radio transmission. We try to find the minimum possible density of nodes. This problem relates to the percolation theory from physics. There is a lower bound on the density below which the nodes of an amorphous computer are split into several disconnected components that cannot communicate with each other. Therefore the density must be above this threshold for the amorphous computer to work properly. Then, we show that the amorphous computer can simulate a computation of a RAM machine in time $O(TDQ \ln(TN/\varepsilon))$, where $T$ is time required on the RAM, $D$ is the diameter of the communication graph, $Q$ is the maximum neighbour count of a node, $N$ is the number of nodes and $\varepsilon$ is the allowed probability of error.

Model 3—the Flying amorphous computer—is introduced in chapter 5. This model exploits mobile nodes. We propose a simple scheme for the node movement, in which all nodes move with the same speed and always stay in the target area. Due to the ongoing node movement we cannot guarantee that a message will be delivered to a target node in any fixed time. Therefore we design a communication protocol requiring a target node to explicitly acknowledge a received message. Should an acknowledgement be not received, message broadcasting is repeated by the sender. Should the delivery of all messages fail, the amorphous computer cannot make any progress in its computation. To avoid such situation, we introduce the property of a *lively flying* amorphous computer: in such a computer the message transmissions cannot fail for an infinite time. For such a case we can prove that a lively flying amorphous computer can simulate a computation of a RAM machine correctly and in a finite time.

The model described in chapter 3 was previously published in [30] and [31]. The model of chapter 4 using sparsely placed nodes was previously published in [32] and [39], and an extended version was published in [40].

The work of chapter 5 on flying amorphous computer was not yet published elsewhere; some preliminary ideas were published in [33].

There is also work [41], [42] applying some of our results about our amorphous computer model for the case of communicating mobile nanomachines. That model is inspired by microbiology and nanotechnology and employs the same communication protocols as described in this work. However, the method used there to show universality is different and is not presented here.

# Chapter 3

# Model 1 of an amorphous computer

An amorphous computer is a machine built from a large number of identical building parts. There are several ways in which an amorphous computer can be organized. They differ in the requirements on the building parts, their distribution, or the effectivity of the computation and communication.

In this chapter we present an approach seeing an amorphous computer as a massively parallel computing system. We will consider a classical model of massively parallel systems—the two-dimensional cellular automaton—and we will design an amorphous computer that can compute like such an automaton.

The obvious difference between a two-dimensional cellular automaton and an amorphous computer is that in an amorphous computer the nodes have a random number of neighbours. Therefore, nodes cannot directly correspond to cells of a cellular automaton. Rather, we organize a group of nearby nodes to act as one virtual cell and we form these virtual cells into a regular square grid. That way, we can map each cell of cellular automaton to one virtual cell on an amorphous computer and for each virtual cell we will also have the required four neighbours.

In this chapter we show that this model can simulate a cellular automaton with a two dimensional square grid and with four neighbours for each cell.

## 3.1 Model 1

**Definition 3.1** A Model 1 of an amorphous computer is a sextuple $C = [N, A, P, r, s, T]$. The model has the following properties:

- The computer consists of $N$ nodes.
- One special node is called the base node.

- Each node is modeled as a point in a square area $A$.
- The positions of nodes are given by a process $P$, which assigns independently to each node a random position in area $A$.
- The nodes operate in rounds; the rounds are synchronized. Node operations within a round need not be synchronized.
- In each round a node receives messages from all of its neighbours (in an arbitrary order), processes these messages and at the end sends one output message (see figure 3.1).
- Each round takes time $T$.

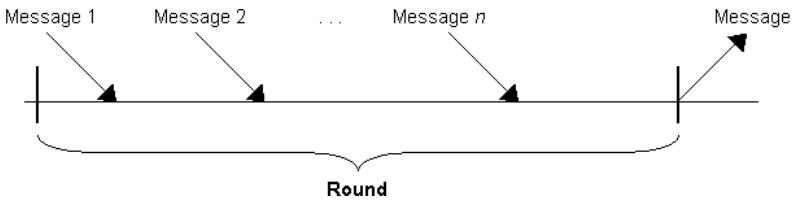



Figure 3.1: Node operation during one round.

Node properties:

- All nodes are equal
- Each node has a fixed number of memory registers of size $s$ bits; initially all registers contain zeros.
- Each node has a fixed number of input and output registers that enable message transmission with neighbours.
- Each node has a control unit that can operate on input/output registers and memory registers. The control unit operates according to a fixed program.

Communication properties:

- Each node has a radio transceiver
- Each radio transceiver operates within a communication radius $r$ on the same channel.
- Nodes in distance less than $r$ are called neighbours and can communicate with each other.

□

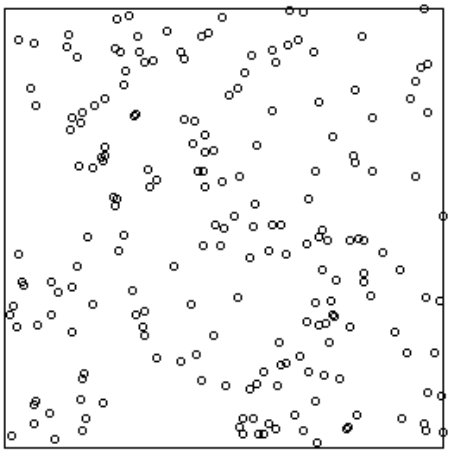


Figure 3.2: An instance of Model 1 of an amorphous computer.

A picture of a random instance of an amorphous computer is shown in figure 3.2.

From computational viewpoint, each node is in fact a finite automaton. We chose to model it as a "small RAM" to make the description consistent with our next models, where finite memory will not be enough. Also, for the description of algorithms it is more convenient to describe them as operations on individual registers than to give a transition function for the whole node state space.

## 3.2 Cellular automaton

A one-dimensional cellular [43] automaton consists of a line of sites. Each site has a state taken from a finite set. The initial configuration of the cellular automaton is the ordered set of states corresponding to all sites. The computation of the cellular automaton starts with the initial configuration and proceeds through synchronized configuration transitions. The state transition of a site is described by a transition rule, which is the same for all sites. The transition rule for a site uses the current state of the site and the current states of a given number of nearest neighbours to the left and to the right of the site and produces a new state. Applying the transition rule to all sites at the same time, the cellular automaton goes from the current configuration to the next configuration.

In a similar way we define the notion of a two-dimensional cellular automaton. The sites are organized in an infinite rectangular grid, each site having one direct neighbour on the left, right, up and down. The transition

rule then uses values of the sites in a fixed local 2D neighbourhood (either four direct neighbours—left, right, up and down—or eight neighbours including the diagonal ones).

For our simulation we place several restrictions on the simulated cellular automaton: The simulated cellular automaton has only finite number of sites (we will call them cells). The cells are organized in a two-dimensional square grid. The transition rule for a cell accesses only four neighbours. Missing neighbours cells at the border of the grid are assumed to be always in a zero state. The cells are initialized with the input configuration. Once started, the computation runs infinitely as a series of configuration transitions.

## 3.3 Necessary node density

The main difficulty in simulating a computation of a cellular automaton on an amorphous computer is the problem of a spatial representation of a regular grid using irregularly placed nodes of an amorphous computer so that the neighbours of cells are located near to each other. Each cell of a cellular automaton requires exactly four neighbours at fixed directions. On an amorphous computer, each node has a random number of nodes in random directions (see figure 3.3). It would be difficult to solve this topological problem if we assigned each simulated cell to some specific nodes. Rather, we tackle this problem by taking a cluster of nodes to simulate a single cell. The cluster roughly corresponds to square area of size $r \times r$, where $r$ is the communication radius. Once the clusters are formed, we will say that the set of clusters forms a virtual grid.

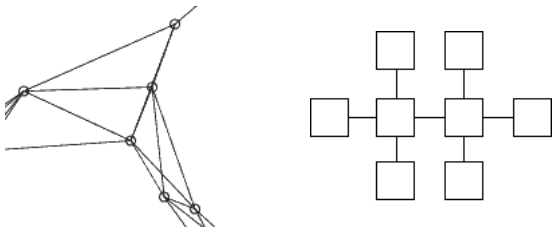


Figure 3.3: The neighbourhood of a node of an amorphous computer is irregular (left), whereas that of a cell of a cellular automaton is regular (right).

In this section we derive conditions that must be met so that the virtual grid can be constructed. In the next section we give the corresponding constructive algorithm.

The aim of the virtual grid construction is to assign the nodes of the amorphous computer to virtual cells of size $r \times r$. The virtual cells should form a regular rectangular grid, in which each virtual cell has four neighbours. The grid construction is successfull only if each virtual cell is nonempty. Moreover, for each two neighbour virtual cells there must be direct communication possible between some pair of nodes, one from each virtual cells. We analyse whether the nodes spread randomly by process $P$ satisfy the above desribed conditions with a high probability and if the simulation is possible.

**Theorem 1** *Let $C = [N, A, P, r, s, T]$ be an amorphous computer whose target square area consists of $M$ squares of size $r \times r$. The probability that some square is not covered by any node or that nodes of one square cannot communicate with any node in some adjacent square is not greater than $5M(1 - \frac{1}{5M})^N$.*

PROOF: A square of size $r \times r$ corresponds to a region whose nodes should form one virtual cells. As the target area can be covered by $M$ such squares, we would like to form $M$ virtual cells. We must make sure that two conditions hold: 1) each square is occupied by at least one node 2) at least one node of a square can communicate with at least one node in neighbouring square. Condition 2 is satisfied if we find two nodes from different squares that are at most $r$ apart.

Let's assume that there is a fine square grid overlaid over the area $A$ ($A$ itself is a square). Let the fine grid have squares of size $r/\sqrt{5} \times r/\sqrt{5}$, see figure 3.4. Assume that there is at least one node in each square of the fine grid. If we take nodes from two neighbouring fine-grid squares, they are at most $r$ apart. This distance is achieved between one corner of the first square and the diagonally opposite corner of the other square.

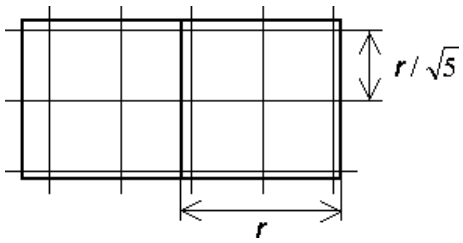



Figure 3.4: At least one full square of a fine grid is contained inside a large square.

If we look at one $r \times r$ square, such a square should contain at least one square of the fine grid. If each fine-grid square contains at least one node, then each square of the larger grid also contains at least one node. Moreover,

there are communication paths through the squares of the fine grid leading to all four neighbour squares of the larger grid.

Now let's compute the probability that all squares of the fine grid are covered. The area of a large grid of $M$ squares can be covered by a fine grid with $5M$ squares. After $N$ nodes are randomly placed on the $5M$ squares, the probability that one square is not covered by any node is $(1 - \frac{1}{5M})^N$. The probability that some of the $5M$ squares is not covered by any node is $5M(1 - \frac{1}{5M})^N$, which is also an upper bound on the probability that some of the squares of the larger grid are not covered or that some nodes from neighbouring squares cannot communicate. □

Note: For $N \to \infty$ the probability that grid construction fails tends to zero. For $N = 5M \ln(2 \cdot 5M)$, the probability that grid construction fails is approximately $1/2$.

## 3.4 Computer set-up

The operation of an amorphous computer has the following phases:

1. Set-up
2. Input entering
3. Computation

The first two phases are controlled by an external operator. The third phase is started by an external operator and then the amorphous computer runs unattended. The external operator operates the amorphous computer through a direct communication channel to one node of the amorphous computer, called the base node. We assume that the base node is located at the top left corner of the target area. After the computation has finished the operator obtains a result through the base node.

In this section we describe the set-up phase, the other phases are described in the following sections. We start by describing data stored in each node.

**Node structure:**

- $i, j$—coordinates of the cell modulo 3
- *left-most*, *right-most*, *top-most* and *bottom-most*—boolean flags denoting nodes on the border of the target area

- *row_set*, *column_set*—boolean flags used for computer set-up
- $k$—the generation count modulo 3
- $q$—state of the cell in the $k$-th generation
- $q'$—previous state of the cell
- $q_{left}, q_{right}, q_{upper}, q_{lower}$—states of the four neighbour cells, or empty if the state is not known

The goal of the computer set-up phase is to prepare so-called well-formed amorphous computer.

**Definition 3.2** We say that an amorphous computer is a *well-formed* $n \times n$ *amorphous computer* if there is at least one node mapped into each of $n \times n$ cells containing the cell's coordinates modulo 3 in registers $i$, $j$ such that the following holds:

- all nodes mapped to the same cell form a connected component,
- for each two neighbouring cells of the $n \times n$ grid there is at least one node in each cell such that the pair of the respective nodes can communicate with each other.

During the the set-up phase we will map the nodes into virtual cells in such a way that each virtual cell will occupy area of size approximately $r \times r$. The resulting virtual cells should be arranged in a regular square grid. Now we describe the set-up process.

In each node the registers $i, j$ that hold the coordinates modulo 3 of the virtual cell to which the node corresponds. The register $i$ holds the row number and $j$ holds the column number. With counting modulo 3 we need only fixed amount of memory in each node irrespective of the total size of the amorphous computer. Since nodes of one virtual cell will send messages only to nodes of directly neighbouring cells, the identification of message sender will be locally unique. Since a node will only communicate with its direct neighbours, counting modulo 3 provides enough information for addressing the correct neighbour. A node can compute addresses of its neighbours as shown in figure 3.5.

In addition to registers $i$ and $j$, each node also has four registers named *left-most*, *right-most*, *top-most* and *bottom-most* holding a boolean value. The *left-most* register will be set to true for all nodes whose virtual cell has no left neighbour, for other nodes it is set to false. Similarly, the registers

$$
\begin{array}{ccc}
 & (i, j-1) \pmod 3 & \\
(i-1, j) \pmod 3 & (i, j) & (i+1, j) \pmod 3 \\
 & (i, j+1) \pmod 3 &
\end{array}
$$

Figure 3.5: Computing addresses of node neighours.

*right-most*, *top-most* and *bottom-most* denote the cells that have no right, up or down neighbour. We will need these registers during simulation of the cellular automaton to ensure that nodes do not wait for messages from non-existing neighbours.

Assume that the target area is a square of size $l \times l$. Let the two sides of the target area be horizontal and the other two vertical. The nodes are initially unaware of their spatial position. However, for the virtual grid formation to be possible we require that at least some nodes know their position.

We require the following to be true: The nodes at distance at most $r$ from the top edge of the target area have the *top-most* register set to true, other nodes have it set to false. The nodes at distance at most $r$ from the left edge of the target area have the *left-most* register set to true, other nodes have it set to false. All nodes have the registers *right-most* and *bottom-most* set to false and registers $i$ and $j$ set to 0.

The virtual grid formation starts with assigning numbers to rows and then continues by assigning numbers to columns. The row number 0 will contain all nodes that are at most distance $r$ from the top edge of the target area. Each next row will consist of the nodes that are at distance approximately $r$ below the preceding row. The column number 0 will contain all nodes that are at most distance $r$ from the left edge of the target area. Each next column will consist of the nodes that are at distance approximately $r$ to the right of the preceding column. The combination of row number and column number will provide the local identification of a cell. Figure 3.6 shows a schematic picture of the intended result. A detailed description of the process follows.

Each node has another two boolean registers named *row_set* and *column_set*, both are initially set to false. When the row formation starts, the external operator issues command (*init_row*). If a node with attribute *top-most* receives this message, it sends the same message to its neighbours and sets its register $i$ to 0 and register *row_set* to true. A node without attribute *top-most* ignores the message. The *row_set* attribute distinguishes the nodes to which row number has already been set. After all nodes of the first row have received the message, the external operator sends message $(row, 1)$. If a node with attribute *row_set* receives this message, it sends the message to

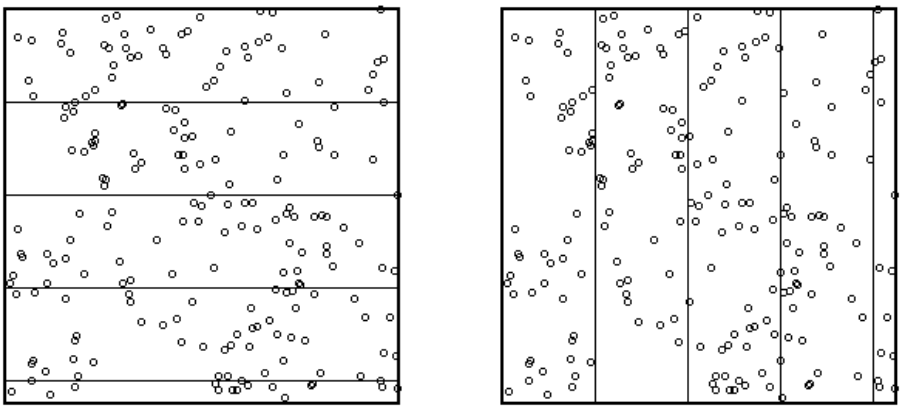


Figure 3.6: Amorphous computer set-up consists of a top-down row formation (on the left) and a left-to-right column formation (on the right).

its neighbours. If a node without *row_set* attribute receives this message, it sets its register $i$ to 1, sets register *row_set* to true and does not resend the message to its neighbours. After all nodes of row 1 have received the message, the operator continues with message $(row, 2)$, and later with $(row, 0)$, counting the rows modulo 3. When the last row should be formed, the operator sends special message $(last\_row, i)$, where $i$ is the number of the row in modulo 3 counting. The $(last\_row, i)$ message works the same as $(row, i)$ message, except that nodes that are newly assigned to this row also set their *bottom-most* attribute to true.

Now we analyse the time complexity of the row formation. After each $(row)$ command is issued, the operator must wait until the message reaches all nodes of the current row. The nodes from the row 0 up to row $i$ are located in an area of size at most $ir \times l$. If we assume that there is a fine grid overlaid over the target area (see proof of Theorem 1), the nodes to which the message must be delivered are spread over $i\sqrt{5} \times l/r\sqrt{5}$ squares of the fine grid. Assume that each square of the fine grid is covered by at least one node. The message is transfered from one square of the fine grid to a neighbouring square in time $T$. The time required till the message is broadcasted to all nodes up to row $i$ is $T(i+l/r)\sqrt{5}$. At most $l/r$ rows can fit into the target area. So, for each row the required time is at most $2Tl/r\sqrt{5}$ and for all rows together it is at most $2Tl^2/r^2\sqrt{5}$.

After the row formation is finished, the external operator starts the column formation. The column formation proceeds analogically to the row formation using messages $(init\_column)$, $(column, i)$ and $(last\_column, i)$. After the column formation is finished, each node has in its registers $i, j$ the coordinates modulo 3 of a simulated cell. Each node can also compute the

coordinates modulo 3 of all four directly neighbouring cells.

**Lemma:** Let $C = [N, A, P, r, s, T]$ be an amorphous computer with randomly spread nodes in area $A$ of size $l \times l$. The time needed for row (column) formation in $A$ is $2Tl^2/r^2\sqrt{5}$.

## 3.5 Experimental validation of computer set-up

To get a picture of how would the result of virtual cells formation look like, we have programmed a computer simulation of the grid formation process. Our program first randomly spreads nodes in a square target area with a prescribed density. We used 10 000 nodes distributed with density 20. We define the density as the average number of neighbours inside the range of a node. The density of 20 corresponds to 6.4 nodes beeing thrown on each $r \times r$ square on average. The row formation and column formation algorithms are simulated with this setting. The result is shown in figure 3.7 in the upper two pictures. The alternating rows are painted using two colours on the left picture, the columns are on the right picture. By combining these two pictures we can see the individual virtual cells. These are shown in the lower left picture. Then we enhanced the edges between points of different colours to see more clearly the structure of the virtual grid. This is shown in the lower right picture.

As we can see, there are no virtual cells missing in the structure. However, due to irregular placement of nodes, the rows or columns formed at some places are very narrow, much less than the expected width $r$. This can be improved if the nodes are distributed with higher density, just as we have analysed in the previous section.

We performed another experiment, this time with 20 000 nodes and density 40. This actually means that the transmission radius of the nodes remains the same, but there are two times as many nodes in the same target area. The pictures resulting from this setting are shown in figure 3.8. We see that the virtual cells generated are much more regular in this setting.

## 3.6 Input entering

After the grid formation phase the nodes contain the local address of the simulated cell but do not yet have any input data. During the input entering phase we transfer the input configuration of the simulated cellular automaton into the amorphous computer. As the external operator can directly communicate only with the base node, which is located in the virtual cell

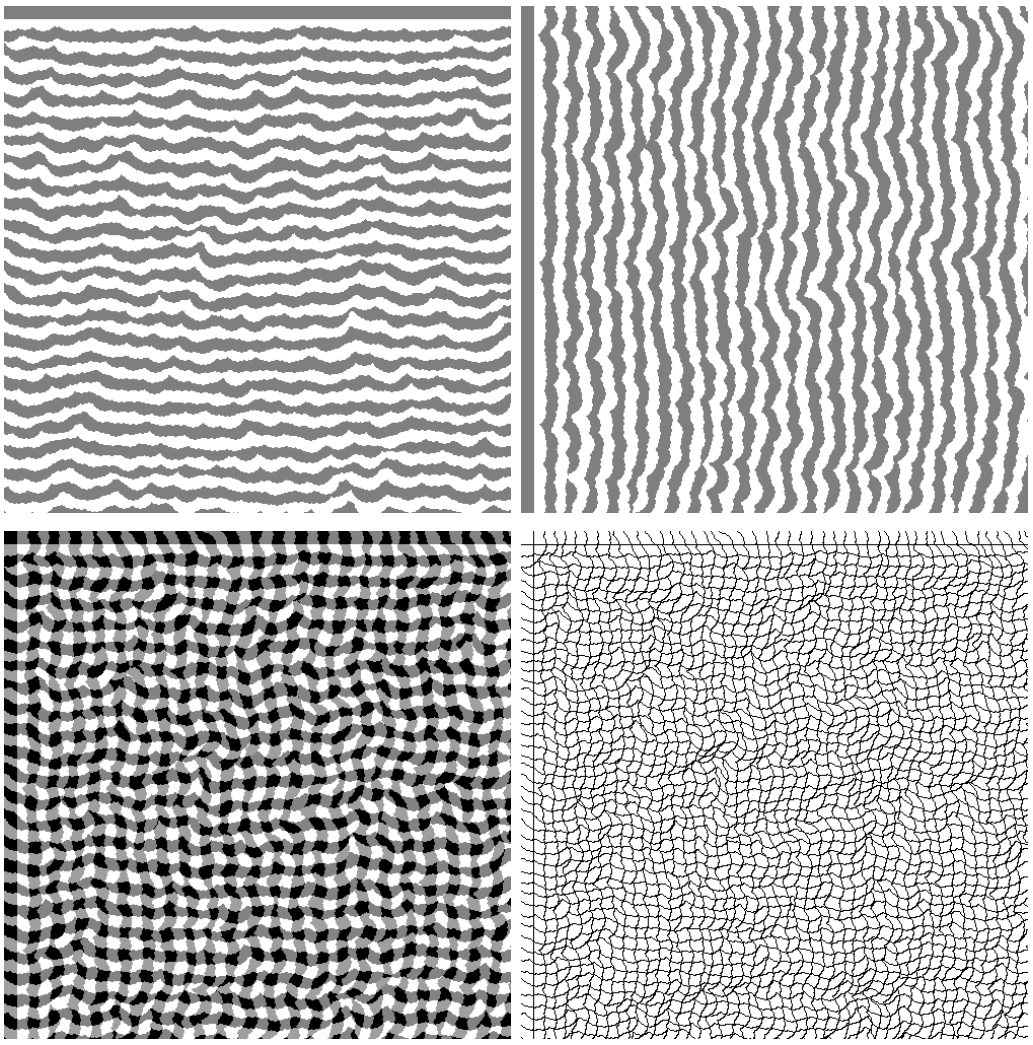


Figure 3.7: Grid formation for nodes dispersed with density 20. The top two pictures show row formation and column formation. The bottom left picture is a combination of both. The bottom right picture is obtained by showing edges of the combined picture.

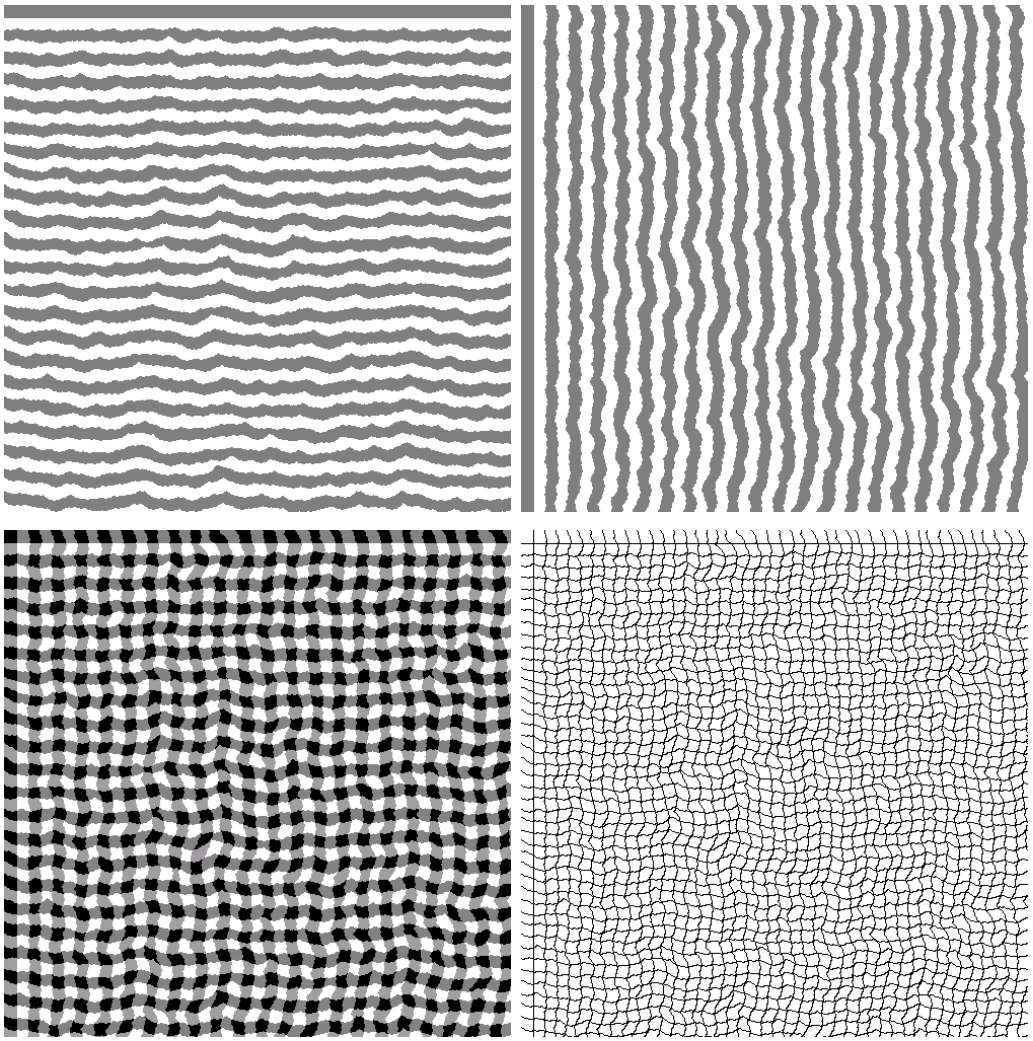


Figure 3.8: Grid formation for nodes dispersed with density 40.

with coordinates (0, 0), the contents of the input configuration will be transfered one cell at a time. Initially, the simulated states of all cells are empty. After the input entering phase finishes, the simulated states of all cells will correspond to the input configuration.

The external operator can use five commands: $(set, x)$, $(move\_right)$, $(move\_right\_end)$, $(move\_down)$ and $(move\_down\_end)$. The $(set, x)$ command sets the state $x$ into all nodes of cell (0, 0). The command $(move\_right)$ makes each cell of the first row to copy the state of its left neighbour. The command $(move\_right\_end)$ marks the end of the $(move\_right)$ command. The command $(move\_down)$ makes all cells copy the state of its upper neighbour. The command $(move\_down\_end)$ marks the end of the $(move\_down)$ command.

The external operator sends the input data in a reversed order, i.e. the contents of the last column last row cell is sent first. By repeatedly issuing the commands $(set, x)$, $(move\_right)$ and $(move\_right\_end)$, the cells of row 0 are filled with the state of the last row to be. Then by issuing command $(move\_down)$, the contents of row 0 is transferred to row 1. After that the operator issues $(move\_down\_end)$ and then he can start issuing the contents of the last but one row to be. This is repeated for all remaining rows.

Now we describe the actions of nodes upon reception of the control messages.

When a node with coordinates (0, 0) receives the $(set, x)$ message it sets its simulated state to $x$ and sends the message to its neighbours. The sending to neighbours is needed so that all nodes with coordinates (0, 0) receive the message. Nodes with other coordinates ignore this message.

Only nodes in row 0 react to $(move\_right)$ messages, other nodes ignore these messages. When a node with coordinates (0, 0) receives the $(move\_right)$ message, it first sends the message to its neighbours and in the next round sends a message in the form $(move\_right, i, j, q)$, where $i, j$ are the coordinates of the node (in this case 0, 0) and $q$ is the current state of the node. If a node receives a message $(move\_right, i', j', q')$ where $i', j'$ correspond to coordinates of its left neighbour, the node first sends the message to its neighbours and in the next round sends a message in the form $(move\_right, i, j, q)$ containing its coordinates and current state, then changes its state to $q'$. After a node has received $(move\_right)$ message to which it reacted, it ignores all subsequent messages until it receives a $(move\_right\_end)$ message.

Only nodes in row 0 react to $(move\_right\_end)$ messages, other nodes ignore these messages. When a node of row 0 receives the $(move\_right\_end)$ message, it sends the message to its neighbours and then ignores all further

(*move_right_end*) messages, until it receives some other type of message.

When a node with coordinates (0, 0) receives the (*move_down*) message it sends the message to its neighbours and in the next round it sends a message in the form (*move_down*, $i$, $j$, $q$), where $i, j$ are the coordinates of the node (in this case 0, 0) and $q$ is the current state of the node. If a node receives a message (*move_down*, $i'$, $j'$, $q'$) where $i', j'$ correspond to coordinates of its upper neighbour, the node first sends the message to its neighbours and in the next round it sends a message (*move_down*, $i$, $j$, $q$) containing its coordinates and current state, then changes its state to $q'$. After a node has received (*move_down*) message to which it reacted, it ignores all subsequent messages until it receives a (*move_down_end*) message.

When a node receives the (*move_down_end*) message, it sends the message to its neighbours and then ignores all further (*move_down_end*) messages, until it receives some other type of message.

We can derive the time complexity similarly as in the computer set-up phase, using the fine grid. The command $(set, x)$ must be spread to all nodes of one cell, which can be done in time $2T\sqrt{5}$. The command (*move_right_end*) must be spread in area $r \times l$, which can be done in time $T(1+l/r)\sqrt{5}$. The command (*move_down_end*) must be spread in area $l \times l$, which can be done in time $2Tl/r\sqrt{5}$. The message sending in (*move_right*) and (*move_down*) commands requires two sending rounds, so it takes at most twice the time for (*move_right_end*) and (*move_down_end*) commands. So, (*move_right*) takes at most time $2T(1 + l/r)\sqrt{5}$ and (*move_down*) takes at most time $4Tl/r\sqrt{5}$.

**Lemma:** Let $C = [N, A, P, r, s, T]$ be an amorphous computer. The time needed to initialize cells in $n$ rows and $m$ columns is at most $mn2T\sqrt{5}+$ $(m-1)n3T(1+l/r)\sqrt{5} + (n-1)6Tl/r\sqrt{5}$.

Proof: For the initialization we use $mn$ times message $(set, x)$, $(m-1)n$ times messages (*move_right*) and (*move_right_end*) and $n-1$ times messages (*move_down*) and (*move_down_end*). □

## 3.7 Simulation algorithm

So far, we have described how the virtual grid is formed on the amorphous computer and how the input configuration is loaded into the amorphous computer. Now we show how the amorphous computer can simulate a computation of a two-dimensional cellular automaton.

We define the generation as the set of the states of all cellular automaton cells at one time. We number the generations produced by a sequence of transition steps by natural numbers. The input data are the generation 0.

By applying the transfer function once we get generation 1, then generation 2, and so on. Each node of the amorphous computer is mapped into one cell of a cellular automaton, and simulates its state. The node holds in its register the number of the current generation modulo 3. In a cellular automaton all cells perform the state transitions synchronously, but in simulation on an amorphous computer different nodes can hold cell values corresponding to different generations. However, a node can be at most one generation ahead of all its neighbours. Under this condition, counting modulo 3 provides enough information to distinguish which nodes are ahead and which are behind.

Each node has one neighbour cell on left, right, top and bottom. A node can compute coordinates modulo 3 of these neighbours. In our case any message will not be transmitted further than to a neighbour cell at distance 2, hence counting modulo 3 is sufficient. If the transition rule of the cellular automaton used larger neighbourhood (e.g. $5 \times 5$), we would use more local addresses, but it would still be a fixed number that fits into a finite memory.

The transition rule is programmed into all nodes. The simulation is started by a control message that is sent from the base node and broadcast to all nodes. We now describe how the simulation proceeds.

**Simulation algorithm:**

Initially, the $i, j$ contain the coordinates of the simulated cell modulo 3, $k$ is set to 0, $q$ is set to the state of the input configuration, $q'$, $q_{left}$, $q_{right}$, $q_{upper}$, and $q_{lower}$ are empty.

While some of the node's registers $q_{left}$, $q_{right}$, $q_{upper}$, and $q_{lower}$ are empty, the node sends in each round a message in the form $(i, j, k, q, q', q_{left}, q_{right}, q_{upper}, q_{lower})$.

During each round the node listens to messages from all its neighbours and records the generation count modulo 3 of the least progressed neighbour.

If a node hears a message from its left, right, upper or lower neighbour that is in the same generation $k$ as the receiving node, the node updates the memory register storing the state of the corresponding neighbour. If a node hears a message from its left, right, upper or lower neighbour that is one generation ahead, it stores the state $q'$ from the message to the register holding the state of the corresponding neighbour.

If a node hears a message from a node with the same coordinates and the same generation $k$ it stores information about the states of the neighbours which are in the message and are not yet in the node's registers.

At the end of a round, a node can proceed to the next generation. However, a node cannot proceed from generation $k$ to $k + 1$ while there is some

neighbour in generation $k-1$. In that case the node ignores the following two rules for proceeding to the next generation.

If a node hears a message from a node with the same coordinates and in generation one ahead, it sets its memory registers according to those contained in the message.

If a node knows the state of all four neighbours, it increases $k$ by one modulo 3, sets $q$ to $q'$, computes the new value of $q$ using the transition function and clears the registers holding the states of the four neighbours.

**Theorem 2** *Let $L$ be a cellular automaton with cells organized in an $n \times n$ grid. Let $C$ be a well-formed $n \times n$ amorphous computer (cf. definition 3.2). Let the virtual cells of $C$ be initialized with the input configuration. Then the simulation algorithm correctly simulates computation of $L$. If one virtual cell and its four neighbour virtual cells are in generation $i$, the cell will be in generation $i+1$ in at most 5 rounds.*

Proof: Assume all nodes of a virtual cell X and all nodes of the four neighbouring virtual cells are at generation $i$. In the next round at least one node of X will hear the status message of the left neighbouring virtual cell. One virtual cell comprises at most $3 \times 3$ subgrid of the fine grid (see proof of theorem 1, where fine grid is introduced). From one node the message is broadcast to all nodes on of this subgrid in at most 4 rounds. So, in 5 rounds all nodes know the state of the left neighbour virtual cell. Similarly for the other three neigbouring virtual cells. The virtual cell $X$ changes to generation $i+1$ in at most 5 rounds.

The correctness of the simulation comes from the fact that a virtual cell goes into the next generation only after it has obtained the states of the four neighbours in the same generation, and the new state is then computed according to the transition rule.

The last thing to verify is that counting generations modulo 3 cannot lead to mistaking generation $k$ for generation $k+3$.

We will show that the following invariant holds: For any node $X$ which is in generation $k$ all the neighbours are in generations $k-1$, $k$, or $k+1$. Initially, the invariant is satisfied since all nodes start at generation 0. Assuming that the invariant initially holds, it could only be violated when node $X$ or some of its neighbours proceed to the next generation. Node $X$ can advance only when there is no node in generation $k-1$, so after the advancement the least progressed neighbours will be at most one generation back. A neighbour of $X$ can advance at most one generation ahead of $X$ so it cannot violate the invariant.

As the invariant holds, there can be no confusion of values $k-1$, $k$ and

$k+1$ in a neighbourhood of any node and no error of computation could be caused by this. □

The simulation does not start in all nodes at once, since the control message must propagate through the network from the base node. However, the control message will reach all nodes in time at most $2\sqrt{5n}$ for an amorphous computer of $n \times n$ virtual cells. Since that time, each new generation is computed in at most 5 rounds.

Measuring the space complexity of the simulation is not straightforward. The number of nodes that we use for the construction of the amorphous computer influences the probability that the amorphous computer can be properly set-up and perform the simulation. If the simulation is possible, the simulation will always be correct. However, if at some place the nodes are not distributed sufficiently densely, the amorphous computer cannot compute correctly. The minimum requirement according to theorem 2 is to cover all cells of the fine grid. So, for the simulation of a cellular automaton with $S = n \times n$ cells we would need at least $5S$ nodes. However, there is very high probability that some fine grid cells will not be covered when using such small number of nodes. As was mentioned in a note below theorem 1, we should use at least $5S \log(10S)$ nodes to assure that the computer set-up is possible with probability at least 1/2. By putting more nodes into the same-sized target area, i. e. by increasing the node density, we can ensure higher probability of success.

The memory requirements for storing data inside each node are not dependent on the total number of nodes $N$, so each node can be a finite automaton.

## 3.8 Possibility of an asynchronous operation

In our Model 1 the nodes work in synchronized rounds and messages are delivered reliably. We could also generalise our model using asynchronously operated nodes, since in the simulation algorithm the nodes have to explicitly synchronise the state transitions anyway. However, real implementations of communication systems usually only guarantee message delivery among synchronized nodes, in which case each node can have its sending timeslot and the messages sent by different nodes do not collide. Thus, to maintain correspondence to possible real implementation we defined our model as a synchronous one.

In the next chapter we present a fully asynchronous model where message delivery need not be reliable any longer.

## 3.9 Conclusion

We have shown that Model 1 of an amorphous computer can simulate a computation of a cellular automaton given that the target area is covered by the nodes with sufficient density. However, the required node density rises with the size of the amorphous computer and for effective memory of $N$ cells we need $O(N \log N)$ nodes. This leads to some wasting of resources for large amorphous computers. On the other hand, the surplus of nodes makes the model robust to nodes that accidentaly stop working.

In our next model we improve on the use of memory resources by designing a model of an amorphous computer that works also for nodes placed with a low density.

# Chapter 4

# Model 2 of an amorphous computer

In this chapter we introduce a slightly different model of an amorphous computer. Whereas the communication issues were severely simplified in Model 1 of an amorphous computer, in Model 2 we will go into more detail with the description of the communication mechanism.

The previous model assumed a flawless message transmission among neighbours, and the transmission was always finished in unit time irrespective of the number of neighbours. However, in real radio networks using one shared channel there can only be one message at a time transmitted to the receiver. Otherwise the transmission fails due to a message collision. As a consequence, should a node receive messages from several senders the transmissions must occur in nonoverlapping unit-time intervals. The required time will be proportional to the number of senders. A transmission scheme minimizing collisions will be achieved making use of randomness. However there is always some probability that a message fails to be delivered. The improved model of amorphous computer captures these characteristics.

## 4.1 Model 2

**Definition 4.1** Model 2 of an amorphous computer is a sextuple $C = [N, A, P, r, s, T]$. The model has the following properties:

- The computer consists of $N$ nodes.
- One special node is called the base node.
- Each node is modeled as a point in a square area $A$.
- The positions of nodes in area $A$ are given by a process $P$, which assigns a random position independently to each node.

- The nodes are not synchronized.
- The nodes operate at nearly the same clock speed. The operation that takes time $T$ on the fastest node takes at most $T(1+\varepsilon_T)$ on the slowest node for a fixed $\varepsilon_T > 0$.

Node properties:

- A node has a fixed number of memory registers of size $s$ bits; initially all registers contain zeros.
- A node has a fixed number of input and output registers that enable message transmission with neighbours.
- A node has one register providing a random number on request.
- A node has a control unit that operates on input/output registers and memory registers. The control unit operates according to a fixed program.

Communication properties:

- Nodes communicate through a shared radio channel.
- Each radio transceiver has a communication radius $r$.
- Nodes at distance less than $r$ from a given node are called neighbours.
- One message transmission takes time $T$.
- A node receives a message if only one of its neighbours sends that message during the transmission time $T$.
- A node cannot receive while sending a message.
- If at the same time instant two or more neighbours of a node are sending a message, a collision occurs and the node receives no message.
- A node cannot detect a collision, the radio receiver cannot distinguish the case when multiple neighbours send messages from the case when no neighbour sends a message.

□

The parameters $N$ and $s$ of the model can in general be chosen independently of each other. However, for the simulation of a RAM machine that we show in this chapter we will always keep these two parameters in relation. Specifically, we need to be able to generate an address for each node, for which we need $\log_2 N$ bits. Therefore we always use $s \geq \log_2 N$. Because of this, we say that the memory requirements of the model are non-uniform.

In a real implementation of the amorphous computer the power required for transmission depends on the size of the communication radius. Therefore, with weak power source, the nodes may be limited to small communication radius.

Smaller communication radius leads to a smaller number of neighbours. Smaller neighbour count also means that less messages need to be delivered and that communication can finish faster. From this perspective, an amorphous computer with small neighbourhood size seems to be a promising model to study.

However, with low neighbourhood size we cannot avoid having "holes" in the target area, i. e., places not covered with nodes. Therefore, we cannot use the spatial distribution of nodes to let them model different cells of a cellular automaton (as in the case of Model 1), because some cells would not have any node assigned to them. To avoid this drawback, we propose another scheme, which assigns unique addresses to the nodes and simulates computation of a RAM machine.

## 4.2 Simulated model – RAM

A random access machine (RAM, cf. [4]) is a computational model consisting of a processing unit, one accumulator and a fixed-size linear array of memory registers. The processing unit works according to a fixed program that is stored in it. Let $Acc$ be the accumulator register, $X_i$ be the register number $i$, $c$ be an arbitrary integer constant and $I_i$ be the $i$-th instruction of the program. The program can use the following types of instructions:

- $Acc := c$
- $Acc := X_i$
- $X_i := Acc$
- $Acc := Acc + X_i$
- $Acc := Acc - X_i$
- $Acc := X_{X_i}$
- $X_{X_i} := Acc$
- Jump to $I_i$ if $Acc > 0$
- Halt

When the Halt instruction is executed the computation terminates.

## 4.3 Communication network properties of interest

When building an amorphous computer according to definition 4.1 we can control the parameters influencing the resulting topology of the network of nodes. We can determine the number of nodes $N$, the size of the target area $A$ and the size of the communication radius $r$. The actual placement of the nodes is done by random process $P$, which is not under our control.

The resulting network topology implies how efectively the amorphous computer can compute and to what extent it can use its available resources. Therefore, the relation between the controlable parameters and the resulting network properties must be considered when designing an amorphous computer.

We assume that the size of the communication radius $r$ depends on the node design and cannot be changed once the node is manufactured. The size of the radius depends on the power of the embedded transceiver and on the battery source of the node. Power consumption rises with increased communication radius (in free space the radio signal is attenuated with the square of $r$, in case of obstacles an exponential attenuation is observed [22]). Therefore, we anticipate that a design with low communication radius will be preferred due to power saving requirements.

Since the communication radius of a node is a parameter which is fixed at the time of its manufacturing, we are only left with two parameters by which we can influence the network topology. These are the number of nodes $N$ and the size of the target area $A$. Regarding the local node density, these two parameters go one against the other. If we increase the number of nodes and at the same time increase the size of the area by the same factor, we will obtain the same density of nodes.

It is the density of nodes which has the major influence on the structure of the underlying communication network. We define the density $d$ of nodes as the average number of nodes per communication neighbourhood area of one node, $d = N\pi r^2/A$. For a uniform random placement of nodes the value $d$ also gives the expected number of neighbours for a node.

The performance of the communication algorithms depends on certain properties of random communication networks. We want to derive the expected and also the extreme values of these properties. The properties of interest are the following ones: the size of the maximal connected component, the maximum number of neighbours, and network graph diameter.

## 4.4 Fraction of accessible nodes

A connected component of a network graph consists of all nodes among which there is either direct communication connection, or indirect connection through several intermediate nodes. In general any network graph may consist of several components. Placing the base node somewhere in the target area, the nodes that are in the same connected component of the communication graph as the base node are the only nodes that can be used for any computation of the amorphous computer at hand. So, we are interested in the fraction of all nodes that is available for the computation.

By choosing the density of node distribution we influence the degree of connectivity and consequently the fraction of nodes accessible from the base node. Starting with very low density, we are at an extreme case where most of the nodes are isolated and have no neighbours. Continually increasing the density, we come to the other extreme, where all nodes belong to a single component. This is optimal as far as the number of accessible nodes is concerned. On the other hand, the high node density slows down the communication and wastes energy.

Changes of network topology similar to ones previously described are studied by percolation theory (cf. [24]). The models investigated in the percolation theory describe, e. g., how a fluid penetrates through a porous material. The accessible nodes in our scenario would then correspond to specific spatial locations which the fluid has reached in the porous material.

The percolation theory usually describes behaviour in an infinite space. The main results of the percolation theory are concentrated around the so-called percolation threshold. The percolation threshold is the critical density above which the spread area of the fluid is infinite. Below the percolation threshold, the fluid can spread only through a finite size area. In our context, this percolation threshold can be interpreted as a point where a possibility for large-scale connection in the network emerges. Estimating the percolation threshold is actually quite difficult to do analytically and usually is done using Monte Carlo simulations. There are few special models where objects are placed on a regular lattice and for these the analytical solution is known.

The percolation threshold for randomly placed nodes in a target area computed with the help of simulation is presented in [20]. The respective threshold in two dimensions occurs at density 4.51. The paper also investigates percolation thresholds in higher dimensions.

The threshold properties are also studied in [23]. Here, the authors investigate a broader range of scenarios. They employ uniform as well as nonuniform node placements, and nodes with the same communication ranges and

also nodes with differing communication ranges.

In our case, when constructing an amorphous computer we deal with the trade-off between low node density and the size of the maximal component. The higher above the percolation threshold is the density, the larger size of the maximal component can be expected. To reveal the above mentioned relation between the fraction of accessible nodes and the node density we performed simulations.

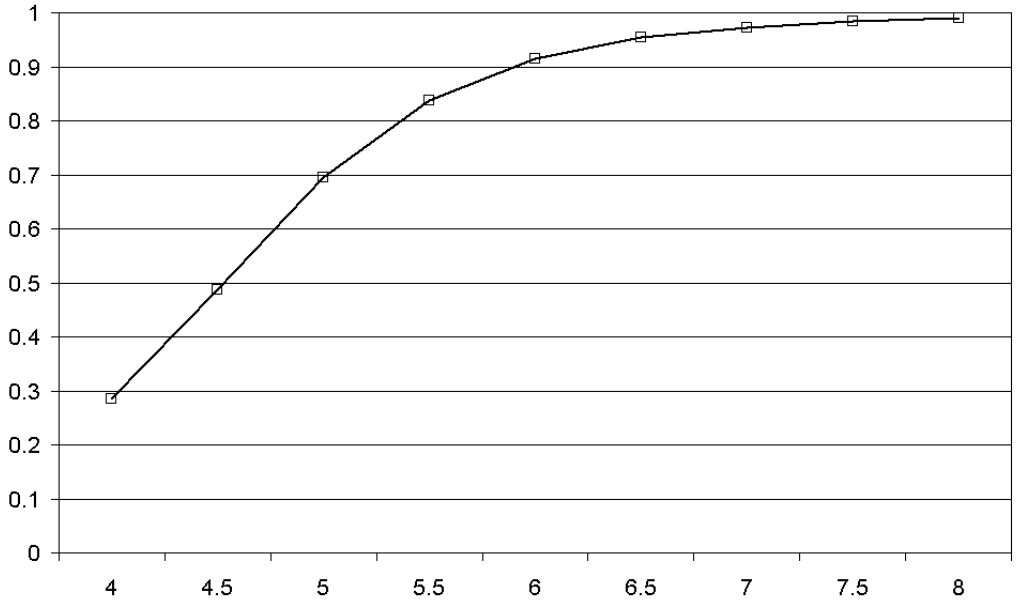



Figure 4.1: Fraction of accessible nodes in dependence on node density.

In our first experiment we randomly threw 400 nodes into a target square area, we placed the base node into the middle of that area and we measured the fraction of nodes that are accessible from the base node. We repeated the same experiment for several node densities. We let the density go from 4 to 8. For each density, we performed 10 000 tests and then calculated the average value. The results are presented in figure 4.1. Based on these result, we decided that density 6 would be the best to be used in Model 2 of an amorphous computer. At this density, the average fraction of accessible nodes is 0.92 of 400 nodes. For lower densities the fraction of accessible nodes quickly drops. Higher densities generally require more transmission energy and do not bring much improvement in terms of the accessible nodes. Hence, density 6 seems to be the lowest density where the fraction of inaccessible nodes seems acceptably low for practical implementation.

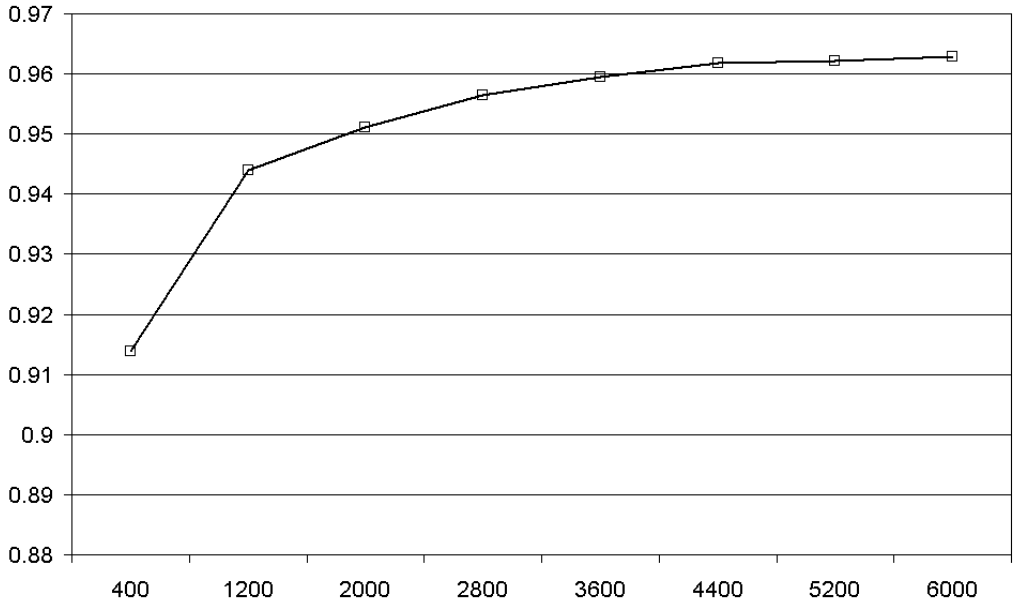



Figure 4.2: Fraction of accessible nodes in dependence on node count. Density is 6.

All tests in the first experiment were done for 400 nodes. We chose this as it is already a large number of nodes and the tests can be performed in a reasonable time.

In our second experiment we studied how the fraction of accessible nodes is affected by increasing the number of nodes. We varied the number of nodes between 400 and 6000. For each number of nodes we performed 10 000 tests and computed the average value. The results are shown in figure 4.2. We see that the fraction of accessible nodes slowly rises but the increase is getting smaller for the large node counts.

Having compared values for different densities and node counts we have learned the trends in the fraction of accessible nodes. However, the average values do not actually tell us what cases we may experience in practice and with what probability. Therefore we set up a third experiment that studied in a higher detail the placement of 400 nodes with density 6. We performed 100 000 random tests. Then we created a histogram showing the frequency of different fractions of accessible nodes. The results are in figure 4.3. The labels on the x axis are the upper bounds of the histogram bins. We observed that there were instances in which the fraction of accessible nodes was very small, lower than 5 %. These tests represented 1 % of all the tests. Then, the medium values of node fractions were very infrequent. Tests with fraction

of accessible nodes lower than 0.55 made in total 2 % of all tests. Finally, the high fractions of accessible nodes happened with increasing frequency. To summarize, we see that low fractions of accessible nodes are encountered with low probability.

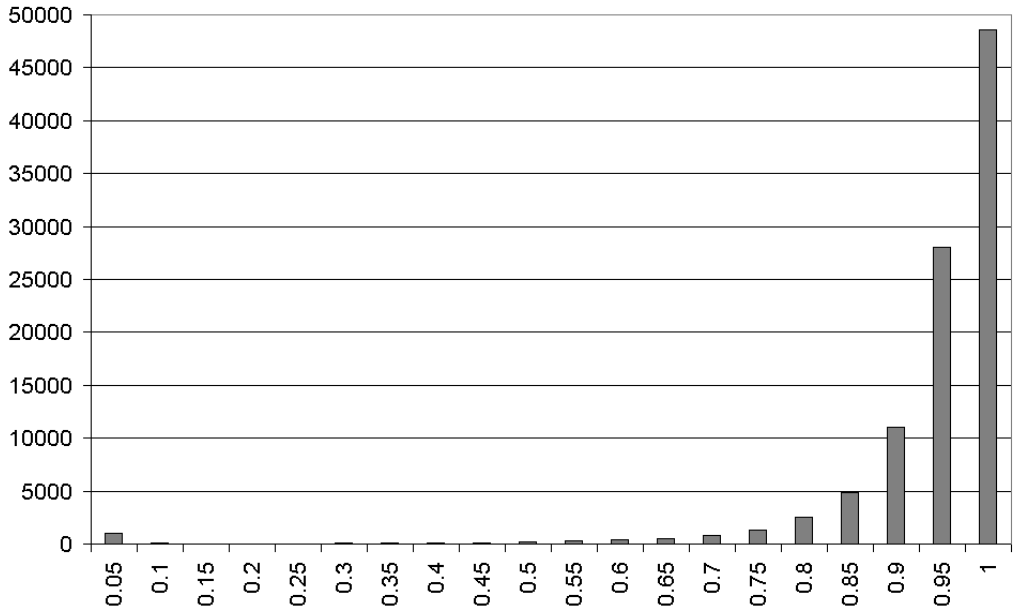



Figure 4.3: Histogram of a fraction of accessible nodes for a base node placed at the center. Density is 6.

In the preceding experiments the base node was always placed in the center of the target area. We were also interested how the fraction of accessible nodes is affected if the base node is placed in a less favourable position. We run the fourth experiment with all conditions the same as in the third experiment, except that this time the base node was placed in the corner of the target area. The results are shown in figure 4.4. We see that the very small fractions of accessible nodes are now encountered with high frequency. The fraction of accessible nodes is lower than 5 % in 46 % of all the tests. When the base node is placed in the corner, lots of random placements end up with impractically small number of accessible nodes.

Seeing this huge difference when the base node is placed either in the center or in the corner of the target area, we are also interested in what happens if the base node's position in the target area is random too. We made another 100 000 random tests and the resulting histogram is shown in figure 4.5. We see that this case is somewhere in between the previous cases. The fraction of accessible nodes is lower than 5 % in 4 % of all tests.

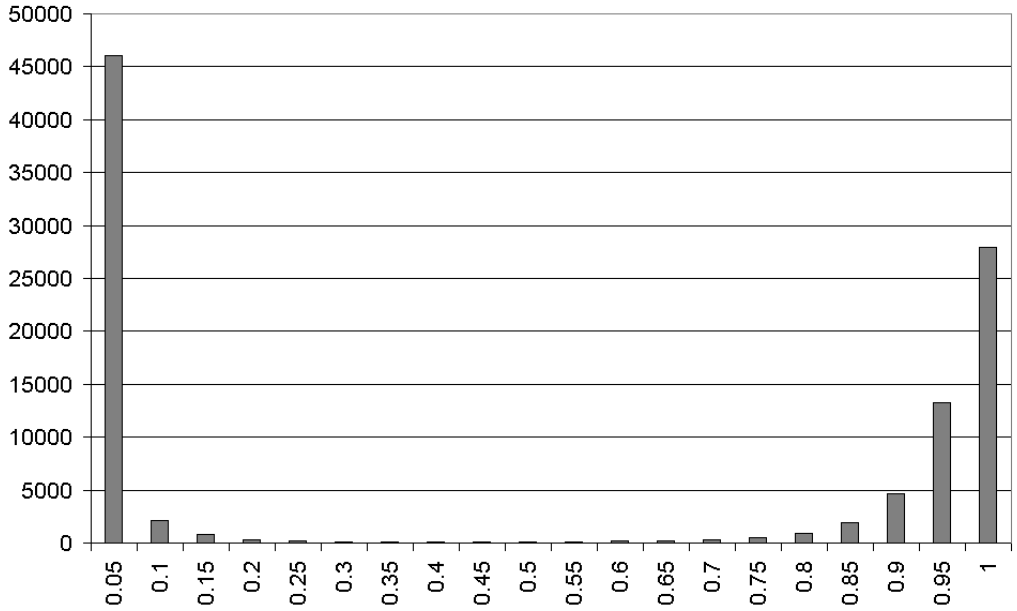



Figure 4.4: Histogram of a fraction of accessible nodes for a base node placed in the corner. Density is 6.

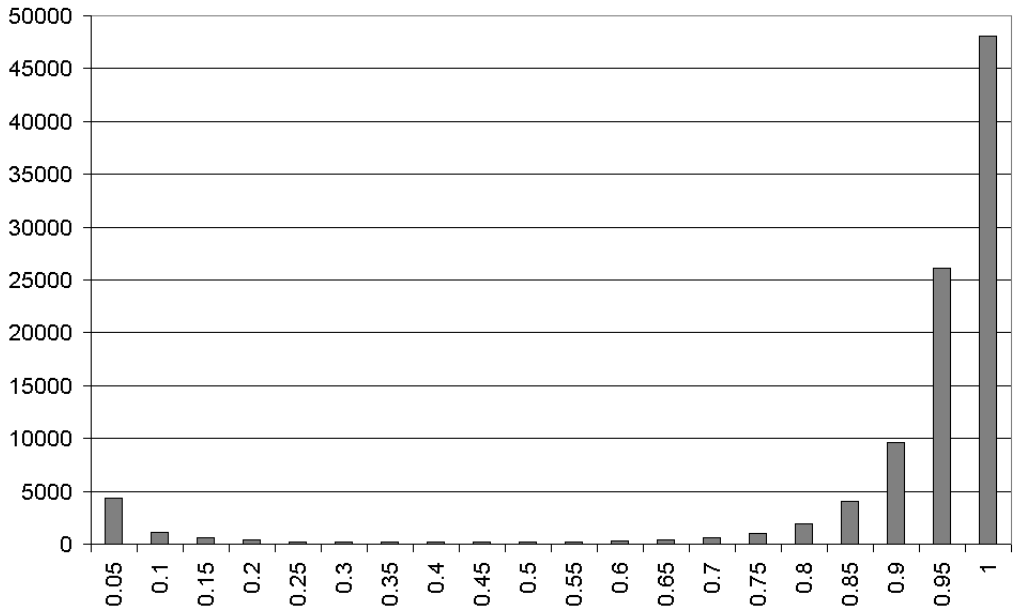



Figure 4.5: Histogram of a fraction of accessible nodes for a base node placed randomly. Density is 6.

To conclude, we propose that, after the nodes of the amorphous computer are randomly dispersed in the target area, the base node should be placed directly to the center of the target area rather than to a random place. This will not increase much the cost of amorphous computer construction, and will yield a higher number of nodes usable for the operation of the amorphous computer.

## 4.5 Number of neighbours

When setting-up an amorphous computer, we must specify the average density of nodes. Obviously, as the nodes are placed by a random process, the actual density at different places will differ. There will be some nodes with no neighbours, while some nodes can have a plenty of neighbours. We assume that for a certain value of average density, there is some upper bound on the number of neighbours for one node. This upper bound has a probabilistic nature, meaning that the actual number of neighbours may still be higher than this bound, but with a low probability.

The number of communication collisions around a node depends on the number of neighbours of that node. The higher the number of nodes that can transmit at the same time, the higher is the collision probability. We must design our communication protocol to work even in the worst case scenario, i.e. when there is the highest expected number of neighbours.

Therefore we will estimate the maximum neighborhood size by applying the techniques known from the solutions of the classical occupancy problem.

**Theorem 3** *Let $C = [N, A, P, r, s, T]$ be an amorphous computer with $N$ nodes randomly uniformly dispersed by process $P$ with density $d$ over a square area $A$, let $Q = \lceil 8 \log N / \log \log N \rceil$. Then for a sufficiently large $N$ the probability that there are more than $12Q$ nodes in any communication neighborhood of a node is less than $4/(dN)$.* [1]

Proof: We start by exactly covering $A$ of size $a$ by $H = h \times h$ squares, with $h \in \mathbf{N}$; the size of each square is chosen so that its size is maximal, but not greater than the area $\pi r^2$ of a communication neighborhood. Then $(h-1)^2 \pi r^2 < a \leq h^2 \pi r^2$ and since $a = N\pi r^2/d$, we get $N/H \leq d$ and for $h \geq 4$, $H/N < 2/d$.

We estimate the probability $p_{=k}$ that in a randomly selected square there will be exactly $k$ nodes for $k$ "not too small" (see in the sequel). Let us consider all sequences of length $N$ over $\{1, 2, \ldots, H\}$ of node "throws" into

[1] In this theorem, the logarithms are to the base 2.

$H$ squares numbered by $1, 2, \ldots H$. There are $H^N$ of such sequences, each of them being equally probable. Consider any $i$, $1 \leq i \leq H$. There are $(H-1)^{N-k}\binom{N}{k}$ sequences containing exactly $k$ occurrences of $i$. Then $p_{=k} = \binom{N}{k}\frac{(H-1)^{N-k}}{H^N} = \binom{N}{k}\frac{1}{H^k}\left(1-\frac{1}{H}\right)^{N-k}$ and the probability that there are at least $k$ nodes in a square is $p_{\geq k} = \sum_{j=k}^{N} p_{=j} = \sum_{j=k}^{N}\binom{N}{j}\left(\frac{1}{H}\right)^j\left(1-\frac{1}{H}\right)^{N-j}$. Using Stirling's approximation $\binom{N}{j} \leq (eN/j)^j$ and upper-bounding the last factor in the last expression by 1 we get $p_{\geq k} \leq \sum_{j=k}^{N}\left(\frac{eN}{jH}\right)^j \leq \sum_{j=k}^{N}\left(\frac{ed}{j}\right)^j \leq \left(\frac{ed}{k}\right)^k \sum_{j=0}^{\infty}\left(\frac{ed}{j}\right)^j \leq \left(\frac{ed}{k}\right)^k \sum_{j=0}^{\infty}\left(\frac{ed}{k}\right)^j$. The latter infinite series converges to $1/(1-ed/k)$ providing $ed < k$. Consider $k$ such that $ed < k/2$; then the sum of the series is at most 2 and for $k \geq (ed)^2$, $p_{\geq k} \leq 2\left(\frac{ed}{k}\right)^k \leq 2.2^{-1/2k\log k}$.

For $k = Q$ we get $p_{\geq k} \leq 2.2^{-\frac{1}{2}\frac{8\log N}{\log\log N}(3+\log\log N-\log\log\log N)} \leq 2/N^2$ (taking into account that for a sufficiently large $N$, $3 + \log\log N - \log\log\log N \geq \frac{1}{2}\log\log N$). It follows that the probability that in any of the $H$ squares there will be at least $Q$ nodes is $2H/N^2 < 4/(dN)$.

Finally, note that for $h \geq 2$ the size of a square is $s > ((h-1)/h)^2\pi r^2 \geq 1/4\pi r^2$. Hence, the area of a communication neighborhood is smaller than the area of four squares. After realizing that the nodes from at most 12 squares can enter a circular neighborhood of area $\pi r^2$ the claim of the theorem follows. □

## 4.6 Network diameter

Communication between two distant nodes of an amorphous computer must proceed through several intermediate hops. Transmission between each two connected hops takes some time. So, the time needed for the message transmission is a factor of hop-to-hop time and the number of hops.

In order to derive the maximum running time of a communication protocol, we need to know the maximum number of hops in a transmission path. For the communication graph, the maximum of the shortest paths between any two nodes is the graph diameter. This is the longest path that must be passed when transmitting a message.

For densely placed nodes, we expect that the number of hops is proportional to the air distance of the two nodes. For nodes placed with low density, there may be some holes, i.e. inaccessible areas, which the transmission path must circumvent. Therefore it is of interest to verify, whether the number of hops (and the diameter size) is also proportional to the air distance in this case.

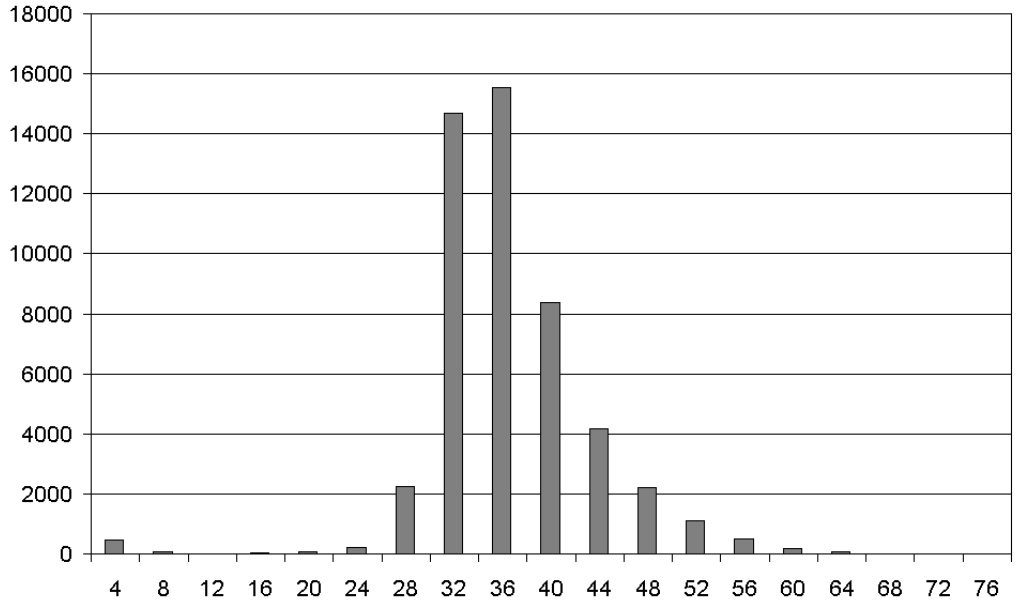



Figure 4.6: Histogram of a graph diameter for 400 nodes.

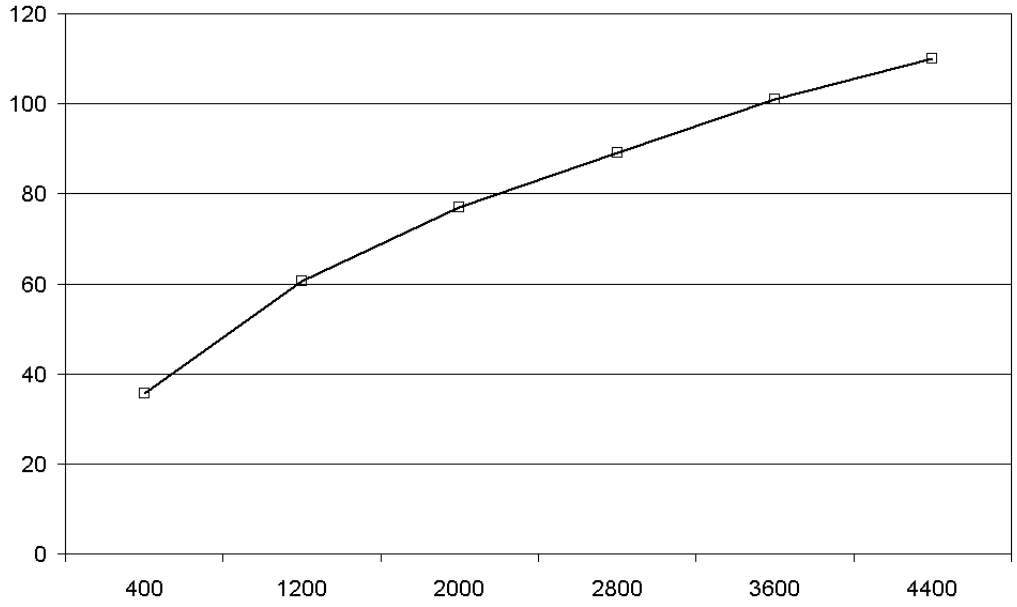



Figure 4.7: Graph diameter for different node counts.

We performed computer experiments to study the diameter of an amorphous computer. We placed randomly 400 nodes with density 6, with the base node placed to the center, and we measured the diameter of the graph component accessible from the base node. We made 50 000 tests. The resulting histogram showing the probabilities of different diameter sizes is in figure 4.6. The numbers on the $x$ axis are the upper bounds of the histogram bins. We see that there are some cases where the graph diameter is very small—in 1 % of the tests the graph diameter was lower or equal to 4. This corresponds well with our analysis of the fraction of accessible nodes. In these cases, the accessible component is just too small. Otherwise, the graph diameter values are spread around a central value—bin 36 in our example.

Next, we studied how the graph diameter changes when more nodes are used for the amorphous computer. We made an experiment in which we varied the number of nodes between 400 and 4400. The node density was 6, the base node placed at the center. We counted only those tests, where the accessible fraction comprised at least 50 % of nodes, otherwise the test was not counted. We made 400 tests for each number of nodes. The results are shown in figure 4.7. We expected a curve of the shape $\sqrt{N}$ and the experiments confirmed that. We performed a linear regression on the measured data which yielded equation $D = 1.6\sqrt{N} + 4$, where $D$ is the accessible component diameter.

Therefore, in deriving the complexity of our algorithms we will assume that $D = O(\sqrt{N})$.

## 4.7 Communication protocols

The nodes can send information to their neighbours via radio. However, if two nodes try to send information to one common neighbour at the same time, due to a message collision no message is received. We will develop a scheme according to which the nodes can transmit so that the collisions are infrequent. This scheme is given for each node by the following protocol Send.

### 4.7.1 Protocol Send

Let $X$ be a node that is about to transmit message $m$ to its neighbours. According to protocol Send the node will go through $k$ periods, each of length $2T$. At the start of each period the node will with probability $p$ randomly decide to send the message. The algorithm ends after the $k$ periods have elapsed. Protocol Send has a parameter $\varepsilon$, which gives the allowed

probability of transmission failure due to a collision. The values of $k$ and $p$ can be determined once $\varepsilon$ is known.

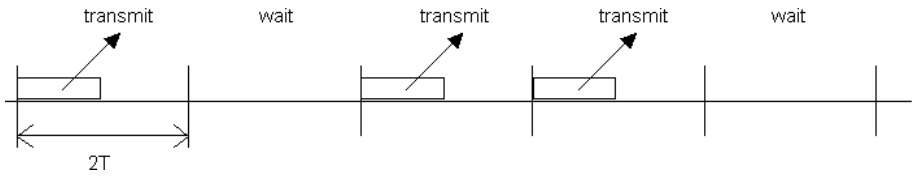



Figure 4.8: Message sending according to protocol Send.

**Theorem 4** *(Send) Let node $X$ be transmitting message $m$ to a neighbour node $Y$ in an amorphous computer $C = [N, A, P, r, s, T]$. Let $\varepsilon$ be the probability of communication failure. Protocol Send($\varepsilon$) delivers message $m$ to node $Y$ in time $2kT$ with probability $\varepsilon$.*

PROOF: Assume that time intervals have the same length on all nodes. Thanks to our choice of the timeslots length, for each timeslot of a given node $X$ there is exactly one corresponding timeslot of some other node $Y$ such that if both nodes send asynchronously in their timeslots, only a single collision will occur. This is so because if $X$ has started its sending at the beginning of its timeslot, $X$'s and $Y$'s sendings overlap if and only if $Y$ had started a sending in a timeslot that was shifted w.r.t. the beginning of $X$'s timeslot by less than $T$ time units in either time direction. The timeslots of length shorter than $2T$ could cause more than a single broadcast collision between the arbitrary pairs of nodes, whereas longer timeslots would delay the communication.

We treat message sendings as independent random events. Message $m$ is correctly received by $Y$ in one timeslot if $X$ is transmitting $m$ (the probability of such event is $p$) and none of $Y$'s neighbors is transmitting (the corresponding probability is $(1-p)^Q$), giving the joint probability $p(1-p)^Q$. The value of $p(1-p)^Q$ is maximized for $p = 1/(Q+1)$. The probability of a failure after $k$ timeslots is $[1 - p(1-p)^Q]^k = \varepsilon$. Hence,

$$k = \frac{\ln \varepsilon}{\ln[1 - p(1-p)^Q]}.$$

The denominator in the latter expression equals $-\sum_{i=1}^{\infty}[p(1-p)^Q]^i/i \leq -p(1-p)^Q = -1/(Q+1)(1+1/Q)^{-Q} \leq -e^{-1}/(Q+1)$ leading to $k = O(Q\ln(1/\varepsilon))$. □

**General case—different clock speeds**

The above described version of protocol Send is simplified for the case when transmission at each node takes time $T$. In a real implementation we may find that different nodes do not have precisely the same clock speed and so the nodes cannot measure exactly the same time intervals. In the definition of Model 2 we introduced the parameter $\varepsilon_T$, measuring the difference of time intervals between the slowest and the fastest operating node. If $\varepsilon_T$ is not 0, the algorithm Send must be modified so that each period has length $T(2+\varepsilon_T)$, where time $T$ is the length of the timeslot in which the node may decide to send a message, and time $T(1+\varepsilon_T)$ is for waiting till the end of the period. Then, the length of the sending interval of the slowest node (being $T(1+\varepsilon_T)$) will fit into the waiting interval of the fastest node. Then the condition in the proof of theorem 4, that a node's transmission can collide with at most one of other node's transmissions, again holds. The time of protocol Send with this modification rises to $(2+\varepsilon_T)kT$. In practical implementation, however, $\varepsilon_T$ must be very small, since otherwise the sender and receiver radios cannot synchronize and communication would not be possible. Therefore, we assume that the time difference between $2kT$ and $(2+\varepsilon_T)kT$ is negligible and in the sequel, for the sake of easier presentation, we consider only the simple version of algorithm Send.

**Discussion of a possible speedup**

The conditions under which the protocol Send works are among the most general possible. It works even in a situation where any node, including the recipient $Y$, is transmitting its own message to its neighbours. This shows, that even if the amorphous computer is initially not organised at all, it is possible to send a reset signal to the computer. The reset signal will be broadcast through the computer (the reset signal is the same for all nodes) and if all nodes correctly react to the reset signal, the amorphous computer will enter a form of organised structure.

In the following sections, we will impose stricter conditions on the node behaviour. Specifically, we will require that before a new message is broadcast, all nodes must be in a quiet state (i.e., not transmitting). Also, when a message is being sent through the computer, it is the same message that is being sent by each node. For the sake of completeness, we note that in that situation we can develop a faster communication. The sending algorithm would be the same, but the values $p$ and $k$ would be different. The cases when the message is spreading least reliably to a node are when either only one neighbouring node is sending the message or when maximum number of neighbouring nodes (i.e., $Q$) are sending the message. If $Q$ nodes concurrently send the same message then the probability that the message is successfully

transfered is $Qp(1-p)^{Q-1}$. If we want to maximize both these extreme cases, we solve the equation $Qp(1-p)^{Q-1} = p$, getting the solution $p = 1 - \frac{1}{(Q-1)\sqrt{Q}}$. The corresponding value of $k$ would be $k = O(Q/\ln Q \cdot \ln(1/\varepsilon))$. Hence, we have obtained an asymptotically faster protocol, indeed. But then we can not reliably reset a computer from an unorganised state using some reset message.

In the rest of the work we only use the first version of protocol Send. We will not use the faster possible protocol, since it is less general and the asymptotic speedup is minor.

It is worth metioning that a problem of message delivery similar to our case has been studied in the seminal paper by Ben-Yehuda, Goldreich and Itai in 1992 [13]. However, that version works only on synchronised networks. Under the same notation as above, the algorithm of Ben-Yehuda et al. runs in time $O((D + \log(N/\varepsilon)) \log N)$. This algorithm is faster than ours, but the assumption of synchronicity (allowing that all nodes can start a required action simultaneously) is a crucial one for its correctness. However, we have no protocol for node synchronization in our Model 2 and developing one would be either very difficult or maybe altogether impossible.

**Memory requirements**

Protocol Send requires memory to store the values of $p$ and $k$. While $p$ is always between 0 and 1, $k$ will rise with smaller error probability $\varepsilon$ asymptotically as $k = O(\ln(1/\varepsilon))$. This would require $O(\ln\ln(1/\varepsilon))$ bits of memory. This function rises very slowly, so it should present no problem in a real implementation of an amorphous computer.

### 4.7.2 Protocol Broadcast

Protocol Send delivers a message to the direct neighbours of a node. We will also need a method to deliver the same message to all nodes of the amorphous computer. For that, we use protocol Broadcast. All nodes of the amorphous computer run according to this protocol.

The protocol Broadcast runs in the following way. Let $X$ be a node and let $m$ be a message received by $X$. Let $m_{prev}$ be the last message received by $X$ before $m$. If $m \neq m_{prev}$ then node $X$ sends the message $m$ using protocol Send. If $m = m_{prev}$ then the message is ignored and $X$ does not send any message.

**Theorem 5** *(Broadcast) Let all nodes initially be in a quiet state, i.e. no node is sending any message. Then, let $X$ be the only node that wants to broadcast message $m$ (different from previous message $m_{prev}$) to all other*

*nodes. Assume that no other node starts a broadcast of a message while broadcast of m is in progress. Let there be N nodes in the network. Let the graph diameter be D. Let $\varepsilon$ be the communication error of protocol Send and $k(\varepsilon)$ be the parameter of protocol Send. Then protocol Broadcast delivers message m to all N nodes and the nodes will be again in a quiet state in time $(D+1)2kT$. The probability that protocol Broadcast fails is $N\varepsilon$.*

PROOF: Call all nodes that receive message $m$ *marked* nodes. Initially, all nodes are unmarked except for node $X$, which is marked. When message $m$ is received by a previously unmarked node, the node becomes marked and starts sending the message using protocol Send for time $2kT$. If $m$ is received by an already marked node, nothing happens, thanks to the condition $m = m_{prev}$ of the broadcast protocol. The protocol is correct, i.e., it ends when all nodes are marked.

The actual sequence and time in which all nodes will be marked varies randomly. However, in the worst case scenario, the marking starts at time 0 with node $X$ and after time $a \cdot 2kT$ all nodes at distance $a$ from $X$ will be marked. As the graph diameter is $D$, after at most time $D2kT$ all nodes will receive the message and after at most time $(D+1)2kT$ the nodes will be again in a quiet state.

The probability that a node does not receive $m$ using protocol Send is $\varepsilon$. Since there are $N$ nodes, the probability that one of them does not receive the message is $N\varepsilon$. □

Note: The proof of the theorem would also work if the same message $m$ were sent from two or more different nodes. The condition, that the message would be delivered to all nodes of the network, would hold. The only difference would be that the broadcasting would terminate sooner. With more source nodes, in time $a \cdot 2kT$ all nodes at distance $a$ from any of the source nodes would have received the message. At time $(D+1)2kT$ all nodes would have received the message and would be in a quiet state again.

We note this property of the Broadcast protocol now, because we will use it in our address assignment protocol in section 4.9.1. It is useful in situations where we need to detect if there are one or more nodes with some property in the amorphous computer. This can be detected by letting all the nodes with that property send the same confirmation message. Since the messages are identical they do not collide.

### 4.7.3 Two-way communication

Protocol Broadcast, introduced in the previous section, has the limitation that only one message can be transmitted through the network at a time. After a message is broadcast through the amorphous computer, the nodes to which the message was addressed should usually send some information back, again using broadcast. However, they should not start broadcasting the reply until the broadcast of the initial message has finished. To avoid overlap of broadcasts of different messages, we develop a two-way communication scheme.

The communication scheme works in the following way: Let *broadcast period* be the maximum time in which a message is broadcasted through the whole network, starting from one node. Assuming that the diameter of the communication graph is known, the broadcast period then has length $(D+1)2kT$.

All nodes always work according to protocol Broadcast, i.e., they retransmit all new messages. The two-way communication proceeds in rounds. Each round can start after all nodes are in a quiet state. One round follows this scheme:

1. The base node sends a request message, and waits for three broadcast periods.

2. Upon receiving a message addressed to a given node, the node waits for one broadcast period and then sends the reply message.

3. The round of the two-way communication scheme ends after three broadcast periods since the start of the round. By that time, the base station has either received a reply from some node (if a reply was sent), or no reply. All nodes are again in a quiet state.

**Theorem 6** *Let $\varepsilon$, $k(\varepsilon)$ be the parameters of protocol Send, $N$ be the number of nodes, $D$ be the diameter of the communication graph and let $B = (D+1)2kT$ be the length of the broadcast period. If the nodes are initially in a quiet state, then one round of two-way communication takes time $3B$ and after that time the nodes are again in a quiet state. During any round there are never two brodcasts of different messages performed simultaneously. The probability of communication error during one round is at most $2N\varepsilon$.*

Proof: Assume that the base node starts sending a message at time 0. By the time $B$ all nodes have received the message and all are in a quiet state. The actual time in which a particular node has received a message is somewhere between 0 and $B$. Because any node first waits for time $B$, its

reply (if any) is sent sometime between $B$ and $2B$, so it does not collide with the first message. At most by time $3B$ the reply arrives to all nodes and all nodes will be in a quiet state. So the schema is correct. The probability of communication error is the sum of the error $N\varepsilon$ during sending the request and the error $N\varepsilon$ during sending the reply. The total communication error of one round of communication is $2N\varepsilon$. □

## 4.8 Operation of the computer

After the nodes are randomly deployed, the computer must be set-up. The setup is controlled by an external operator, who is connected to the base node. The setup consists of two phases. In the first one, we generate unique address for each node. In the second one, we fill the initial data to nodes.

Then, the external operator starts the computation, and after that he or she can disconnect from the computer. The computation runs unattended, controlled by the base station.

At some later time, the external operator can again reconnect to the computer, to find out if the computation has finished and to read the output value.

## 4.9 Computer set-up

### 4.9.1 Address Assignment

During the address assignment phase we want to assign each node a unique address in the range 0 to $N - 1$.

The idea of the address assignment algorithm is the following. Initially, no node has an assigned address. The algorithm proceeds in rounds. In each round one address will be assigned to one node. At the start of the round we take all nodes to which no address has been yet assigned and mark the nodes as active. Then we split the active nodes randomly into two groups 0 and 1, where one of the groups may be empty. If the group 0 is nonempty, we let all the nodes of group 1 become inactive. If the group 0 is empty, we implicitly select group 1. After each splitting the number of active nodes can decrease, however, there will always be at least one active node (see figure 4.9). The splitting procedure is performed several times so that unique node is selected with a high probability. After the splittings are finished, we assign next available address to the active node. Then, if there are still some addresses to be assigned, the algorithm continues with a next round.

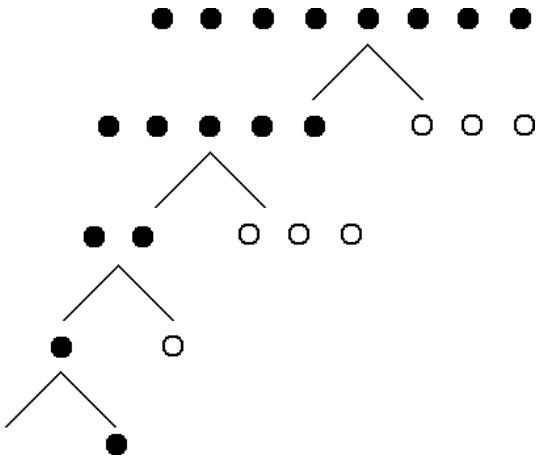


Figure 4.9: Selecting one node by iterative splitting.

The algorithm will be given as a set of instructions performed by the base node and another set of instructions performed by the other nodes. In addition, all nodes work according to Broadcast protocol, i.e., they retransmit each message.

Variables at the base node:

- $m$—the necessary number of splitting rounds, a parameter set by the operator (cf. Theorem 7)
- $next\_address$, initially 0

The instructions for the base node:

1. Send message ($Init\_split$) and wait for three broadcast periods.
2. If no reply is received the algorithm ends, because there are no more nodes without an address.
3. Repeat $m$ times:
   Send message ($Split$) and wait for three broadcast periods.
   If a reply ($Group\_nonempty$) is received, send message ($Successful\_split$) and wait for one broadcast period.
4. next_address = next_address+1
5. Send message ($Assign, next\_address$).
6. Go to step 1.

Variables in other nodes:

- *address*, initially empty
- *active*—boolean
- *group*—has values 0 or 1

The instructions for other nodes:

- On message ($Init\_split$):
  If the node has no address assigned, then the node changes to active and after one broadcast period sends message ($Ready$).
- On message ($Split$):
  If the node is active, the node randomly assigns itself to group 0 or 1. If the new group is 0, the node sends message ($Group\_nonempty$).
- On message ($Successful\_split$):
  If the node is active and in group 1, the node changes to inactive
- On message $(Assign, a)$:
  If the node is active, it assigns itself address $a$ and becomes inactive.

**Theorem 7** *Let the address assignment algorithm run with parameter $m$ on an amorphous computer with $N$ connected nodes. The probability that some address is assigned to two or more nodes is at most $N^2/2^m$.*

Proof: When assigning one address, there is some sequence of group 0 or 1 selections that leads to the finally picked node. The probability that some other node made exactly the same sequence of group 0 or 1 selections is $N/2^m$. There are at most $N$ addresses assigned. The probability that there is some address that was assigned to more than one node is at most $N^2/2^m$. □

**Theorem 8** *Let $m$ be the input parameter of the address assignment algorithm. Let $\varepsilon$ be the parameter of protocol Send. Let $B$ be the broadcast period. Let an amorphous computer $C$ have $N$ connected nodes. The address assignment algorithm will run on $C$ in time $O(NmB)$ and the probability of communication error during the algorithm run is $(3m+3)N^2\varepsilon + 2N\varepsilon$.*

Proof: Each splitting iteration takes time $O(B)$. There are $m$ iterations, and $N$ addresses are assigned in total, so the time complexity is $O(NmB)$.

During each splitting iteration, the maximum communication error for messages $Split$, $Group\_nonempty$ and $Successful\_split$ is together $3N\varepsilon$. For

all $N$ addresses and $m$ iterations the error for splitting messages is $3mN^2\varepsilon$. The error for $N+1$ $Init\_split$ messages is $(N+1)2N\varepsilon$. The error for $N$ address assignments is $N^2e$. In total, the communication error is $(3m+3)N^2\varepsilon+2N\varepsilon$. □

The address assignment fails either if there is a communication error or if two nodes are assigned the same address. The total probability that the address assignment algorithm fails is $N^2/2^m + (3m+3)N^2\varepsilon + 2N\varepsilon$.

### 4.9.2 Data Input

During the data input phase all input data for the computation are stored to the memory registers of the individual nodes.

The data input is controlled by the base node. For each memory address the base node issues message $(Write, i, x)$, where $i$ is the address of the register and $x$ is the value to be stored in that register. The base node then waits for three broadcast periods and then continues with the next memory address. The operation of a node after receiving the $(Write, i, x)$ message is described in the Simulation section.

For $N$ addresses the time needed for data input is $3NB$, where $B$ is the broadcast period. If $\varepsilon$ is the parameter of protocol Send then the maximum communication error of data input is $3N^2\varepsilon$.

## 4.10 Simulation

The simulation is controlled by the base node. During the simulation the base node performs one by one the instructions of its stored program. The base node also holds the value of the instruction counter and accumulator. Initially the instruction counter points to the instruction number 1, and the contents of the accumulator is zero. All memory registers are initialised with the input values. Simulation of one instruction may require zero, one or two reads or writes of the values of the memory registers. Two memory accesses are required for indirect read and indirect write instructions. Each read or write operation is simulated on the amorphous computer using one round of two-way communication. The following messages will be sent:

- If the base node simulates memory read, it sends the message $(Read, i)$. If the base node simulates memory write, it sends message $(Write, i, x)$, where $i$ is the address of the register and $x$ is the value. The base node then waits for three broadcast periods

- If a node with address $i$ receives message $(Read, i)$, then after one broadcast period the node sends message $(Value, x)$ where x is the node's value. If a node with address $i$ receives message $(Write, i, x)$ it stores the value $x$ and after one broadcast period the node sends message $(Written)$.

- If the simulated operation was memory read, the base node receives the message $(Value, x)$ and uses $x$ as the result of the read operation.

**Theorem 9** *Let $R$ be a unit cost RAM with bounded registers that can contain at most $q$-bit numbers. Let $C$ be a computation of $R$ that takes time $T_1$ and space $S_1$. Let $AC$ be an amorphous computer with $N \geq S_1$ nodes in a connected component, to which addresses 0 to $N-1$ have been assigned. Let the size of $AC$'s registers be $s$ bits, $s \geq q, 2^s \geq N$. Let $\varepsilon$ be the communication error of protocol Send and $B$ be the length of the broadcast period. The computation $C$ can be simulated by $AC$ in time $6T_1B$ with maximum error $4T_1N\varepsilon$.*

PROOF: First we verify that conditions for correct operation of protocol Broadcast are satisfied. Protocol Broadcast cannot broadcast a message identical to the immediately preceding one. In the simulation algorithm, messages $Read/Write$ alternate with messages $Value/Written$, so this condition is preserved. The other condition is that two broadcasts cannot take place at the same time. This is ensured by using the two-way communication scheme.

The correctness of the output value of the computation can be deduced from the straightforward simulation of every instruction of the RAM.

An error is possible only at the level of protocol Send. While simulating one instruction, at most four messages will be broadcast. After $T_1$ instructions, the maximum error is $4T_1N\varepsilon$.

One round of a two-way communication takes time $3B$. Simulation of each instruction may require at most two rounds of a two-way communication. After $T_1$ instructions, the total time is $6T_1B$. □

**Theorem 10** *Let $\varepsilon_1$ be the given error probability. Let $R$ be a unit cost RAM with bounded registers that can contain at most $q$-bit numbers. Let $C$ be a computation of $R$ that takes time $T_1$ and space $S_1$. Let $AC$ be an amorphous computer with $N \geq S_1$ nodes in a connected component to which addresses 0 to $N-1$ have been assigned. Let the size of $AC$'s registers be $s$ bits, $s \geq q, 2^s \geq N$. Let $D$ be the diameter and $Q$ be the maximum degree of the communication graph. Then $C$ can be simulated by $AC$ in time $O(T_1DQ\ln(T_1N/\varepsilon_1))$ with error probability $\varepsilon_1$.*

PROOF: If the error probability of algorithm Send is $\varepsilon_1/(4T_1N)$ then the total error probability of the simulation is $\varepsilon_1$. The time complexity of algorithm Send is $O(Q\ln(T_1N/\varepsilon_1))$. This leads to the time complexity of the simulation $O(T_1DQ\ln(T_1N/\varepsilon_1))$. □

## 4.11 Conclusion

In this chapter we introduced Model 2 which is substantially different from the previous Model 1. This change requires a complete redesign of the way in which this amorphous computer operates.

While Model 1 computed in a parallel mode by simulating a cellular automaton, this is not possible with Model 2. Thus, for tasks where an efficient parallel algorithm is known, Model 2 is not as efficient as the previous model. The main reason why the same approach would not work is that nodes of Model 2 cover the target area sparsely and there can be holes or nodes that are inaccessible. Therefore, we needed to organise the computation in a way that the data used during the computation need not be stored in a specific spatial location. This is why we have chosen to simulate a RAM, in which the registers are directly adressed and therefore need not be spatially organised in any particular way.

However, from the memory effectiveness point of view, Model 2 is more effective because no data are duplicated in multiple nodes.

Another difference from Model 1 is in the memory size of a single node. Model 1 used nodes with constant memory, while in the present model the memory must rise with $O(\log N)$, so as to be able to store addresses in range 0 to $N-1$. This should not pose any problem for a practical realization, since already with 32 or 64 bits we can address an enormous number of nodes.

Note that the nodes in Model 2 operate asynchronously, which for concrete implementation is a more realistic assumption than the synchronous rounds of Model 1.

Thanks to the use of addresses it is not important where a node is located as long as it is connected to the network. Thus, if for some reason a node moved to a different place in the target area, still being in the same component as the base node, the computation could continue. We will use this property when developing our next model, called the flying amorphous computer, where nodes would be changing locations all the time.

# Chapter 5

# Model 3: Flying amorphous computer

In this chapter we present a substantial extension to our Model 2 of an amorphous computer. This new model makes use of moving nodes and therefore we call it a flying amorphous computer. In reality, one could assume that the nodes of a flying amorphous computer would be attached to some objects that are randomly moving in some confined space. These moving objects could be, e. g., herd of living animals walking on ground (e.g. cows, horses, chicken), birds flying in the air, or fish. Alternatively, these objects could be some autonomous group of moving robots, e.g., miniature walkers or helicopters designed to reach some hardly accessible buildings or dangerous locations. Obviously, the actual nature of the moving objects and the task they accomplish will most surely affect the moving patterns of the nodes. Nevertheless, in our modelling we do not want to run into difficulties of analysing what is the most likely moving pattern of animals or robots. Rather, in our study of flying amorphous computers we propose a simple model of node movement that is based on few simple rules.

The internal structure and working of the nodes of the flying amorphous computer is the same as it was in the Model 2 of the amorphous computer. Similarly, the nodes are initially placed randomly with uniform density in a bounded target area. We assume the target area has a square shape. After the nodes are placed, they constantly move in that area. In our simple movement model each node is assigned a non-zero random vector and the node is moving with constant speed in the direction of that vector. The speed is the same for all nodes. We have to ensure that the nodes remain in the target area. Therefore, we make the edges of the target area to bounce the nodes. If a node touches an edge of the target area, it is bounced like a billiard ball back into the target area (see figure 5.1). Note that there are several possibilities for choosing a new movement vector of a node hitting the

edge. The mirror-like bouncing method has the advantage that it maintains the same average density of nodes in all parts of the target area.

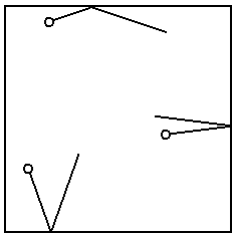


Figure 5.1: Nodes bounce off the target area edges.

In the case of a static amorphous computer, after the nodes were deployed, some of them were accessible from the base node, while others were inaccessible. In the case of a flying amorphous computer, the the position of a node is dynamically changing. At any time instant, there is a subset of the nodes that is accessible from the base node through intermediate nodes. However, as time goes, some nodes move away from this subset while other nodes enter this subset. These changes proceed even while a communication protocol is progressing, which means that the nodes that finally receive a message are not necessarily the nodes that would be accessible when a message transmission started. All this makes it very complicated to analyse the probability that a message will enventually be delivered from one node to another one, or the probability that a message broadcasted by one node will be received by all other nodes.

However, the algorithm that we have used for communication on the static amorphous computer (cf. section 4.7) will, with small changes, also work on a flying amorphous computer, as it can be verified by a computer simulation. The idea of the there-used protocol Send was for each node to repeatedly send a message $k$ times with probability $p$. By increasing the number $k$ the probability of communication error could be made arbitrarily small. On the flying amorphous computer we also have the communication errors caused by transmission interferences and these errors can be made smaller by increasing the repetition count $k$. But now we also have an other type of errors caused by node inaccessibility.

A node that does not have any neighbour cannot receive any message. However, we can assume that a node can remain isolated only for a limited time. As a node moves, it sooner or later meets some other node in the target area. By increasing $k$ (the number of repetitions) we provide a larger time window in which the inaccessible node may again become accesssible. Perhaps by using a very large value of $k$ the error due to node inaccessibility

could be made arbitrarily small. However, due to the complex nature of the system at hand we did not succeed in analytic derivation of the relation between value the value of $k$ and communication error $\varepsilon$. Therefore, to find a reasonable value of $k$ for an amorphous computer of given size we rely on a computer simulation.

Now we give the full definition of a flying amorphous computer.

**Definition 5.1** A model of a flying amorphous computer is a septuple $C = [N, A, P, r, s, T, v]$. The model has the following properties (same as in Model 2):

- The computer consists of $N$ nodes.
- One special node is called the base node.
- Each node is modeled as a point in a square area $A$.
- The initial positions of nodes in area $A$ are given by a process $P$, which assigns a random position independently to each node.
- The nodes are not synchronized.
- The nodes operate at nearly the same clock speed. The operation that takes time $T$ on the fastest node takes at most $T(1+\varepsilon_T)$ on the slowest node for a fixed $\varepsilon_T > 0$.

Node properties (same as in Model 2):

- A node has a fixed number of memory registers of size $s$ bits; initially all registers contain zeros.
- A node has a fixed number of input and output registers that enable message transmission with neighbours.
- A node has one register providing a random number on request.
- A node has a control unit that can operate on input/output registers and memory registers. The control unit operates according to a fixed program.

Movement:

- Initially, a non-zero random direction vector $u$ is assigned to each node. Thereafter, each node moves with a constant speed $v$ in the direction of its vector $u$.
- A node does not know and cannot influence its direction vector.
- If a node is about to leave the target area $A$ then its direction vector is mirrored as if a billiard ball would bounce off a wall.

Communication properties (same as in Model 2):

- Nodes communicate through a shared radio channel.
- Each radio transceiver has a communication radius $r$.
- Nodes at distance less than $r$ from a given node are called neighbours.
- One message transmission takes time $T$.
- A node receives a message if only one of its neighbours sends that message during transmission time $T$.
- A node cannot receive while it is sending a message.
- If at the same time two or more neighbours of a node are sending a message, a collision occurs and the node receives no message.
- A node cannot detect a collision, the radio receiver cannot distinguish the case when multiple neighbours send messages from the case when no neighbour sends a message.

□

Note that our model of a flying amorphous computer is a generalization of Model 2, indeed.

## 5.1 Broadcasting protocol

We start the description of a flying amorphous computer operation with its communication protocol. The communication protocol of a flying amorphous computer will have several features similar to previously described protocols Send and Broadcast (cf. section 4.7).

Protocol Broadcast that we used for the static amorphous computer allowed only one message to be transferred through the network at a time. The time for the broadcast operation depended on the network diameter $D$ and the parameters of the protocol Send.

The broadcasting protocol on the flying amorphous computer will also allow only one message to be transferred through the network at a time. For the flying amorphous computer, however, we cannot define anything like the network diameter. The network as a whole can be disconnected at all times but still the links that temporarily form and then disappear may allow the message to be passed through one node to any other node. That is, the original communication algorithm may work on a flying amorphous computer even though no nodes are permanently connected to the node which started the transmission. However, the temporary and arbitrary emergence of paths

could lead to a situation where the running time of the old Broadcast protocol would be unexpectedly long. In the worst, but very improbable case, the starting node would be connected to only one other node, later that node would connect to another one node and the message could travel through the network visiting all nodes one by one. In that case the running time would be dependent on $N$ rather than $\sqrt{N}$, where the latter corresponds to the diameter of the static amorphous computer and would be the expected case.

To prevent a large difference between the expected and the worst running time of the broadcasting protocol we develop a new protocol for the flying amorphous computer. We call it protocol Broadcast2. The main difference from the previous algorithm Broadcast is that the Broadcast2 protocol will, as a part of the message, also send the number of broadcast steps remaining to the end of the broadcasting. With this modification all nodes stop broadcasting at the same time (up to a difference caused by imprecise clock speeds of the nodes). Also, we let all nodes to broadcast all the time, until the protocol finishes, rather than stop the broadcast after making several sending attempts. On the static amorphous computer the neighbourhood of each node was static and therefore, after a fixed number of sending attempts, all neighbours have received the message with a high probability and no further attempts were necessary. However, in a flying amorphous computer the neighbourhood of a node is continually changing and therefore it helps if each node continues with the sending attempts all the time while the broadcasting protocol runs.

```
Broadcast2(l, m)
begin
    disable receiving messages;
    while l > 0 do
        if rand < p then            /* Send with probability p */
            send message [l − 1, m];
        end
        wait 2T;
        l = l − 2;
    end
    enable receiving messages;
end

On message [l, m]
begin
    Broadcast2(l, m);
end
```

**Algorithm 1**: Protocol Broadcast2.

Algorithm 1 shows the code of the Broadcast2() procedure and how a node handles an incoming message. The message handling is simple: Every node hearing a message $[l, m]$ of protocol Broadcast2 splits the message into two parts, $l$—the time to continue the broadcasting, and $m$—the data part of the message. The node then starts broadcasting by running procedure Broadcast2().

While performing Broadcast2() procedure the node blocks reception of any new messages. Thanks to this, a node does not react to its own message when it is sent back to it through some of its neighbours. When Broadcast2() procedure ends, the reception of messages is again allowed.

Procedure Broadcast2() has an implicit parameter $p$, whose value is the same for all nodes. This parameter controls the rate at which nodes attempt sending messages. The optimal value of this parameter depends on the number of node's neighbours. Since the number of node's neighbours on a flying amorphous computer varies, the optimal value of $p$ is best selected with aid of computer simulation.

The working of Procedure Broadcast2() is similar to that of algorithm Send (see 4.7). At intervals of length $2T$ the node periodically makes attempts at sending the messages. At each attempt the message is sent with probability $p$. This is repeated until the runnning time of the broadcasting $l$ goes to zero.

**Theorem 11** *Let all nodes be in a quiet state. Let one node send a message using protocol Broadcast2 with parameters $l$ and $m$. After time $lT(1 + \varepsilon_T)$ all nodes will again be in a quiet state.*

Proof: Let's consider the first node that starts the broadcast. The node decrements its variable $l$ by 2 each multiple of time $2T$. The node will become quiet in time $lT$ according to that node's time measurement. However, the actual time it may take is $lT(1 + \varepsilon_T)$, due to expected inaccuracies of time measurement in individual nodes. Every other node that receives the broadcast message also receives the updated value $l$. The value sent in the message is $l - 1$, which is the value of parameter $l$ on the originating node after the message transmission has finished. So, all nodes that take part in the broadcast have approximately the same value of parameter $l$ and all differences are caused by the imprecise time measurement in individual nodes. The running time for the slowest nodes is bounded by $lT(1 + \varepsilon_T)$. □

After a node starts a broadcast its message starts spreading around that node roughly in a shape of circles. The spreading of a message is similar to models of infection spreading. See figure 5.2 showing a computer simulation of message broadcasting on a flying amorphous computer.

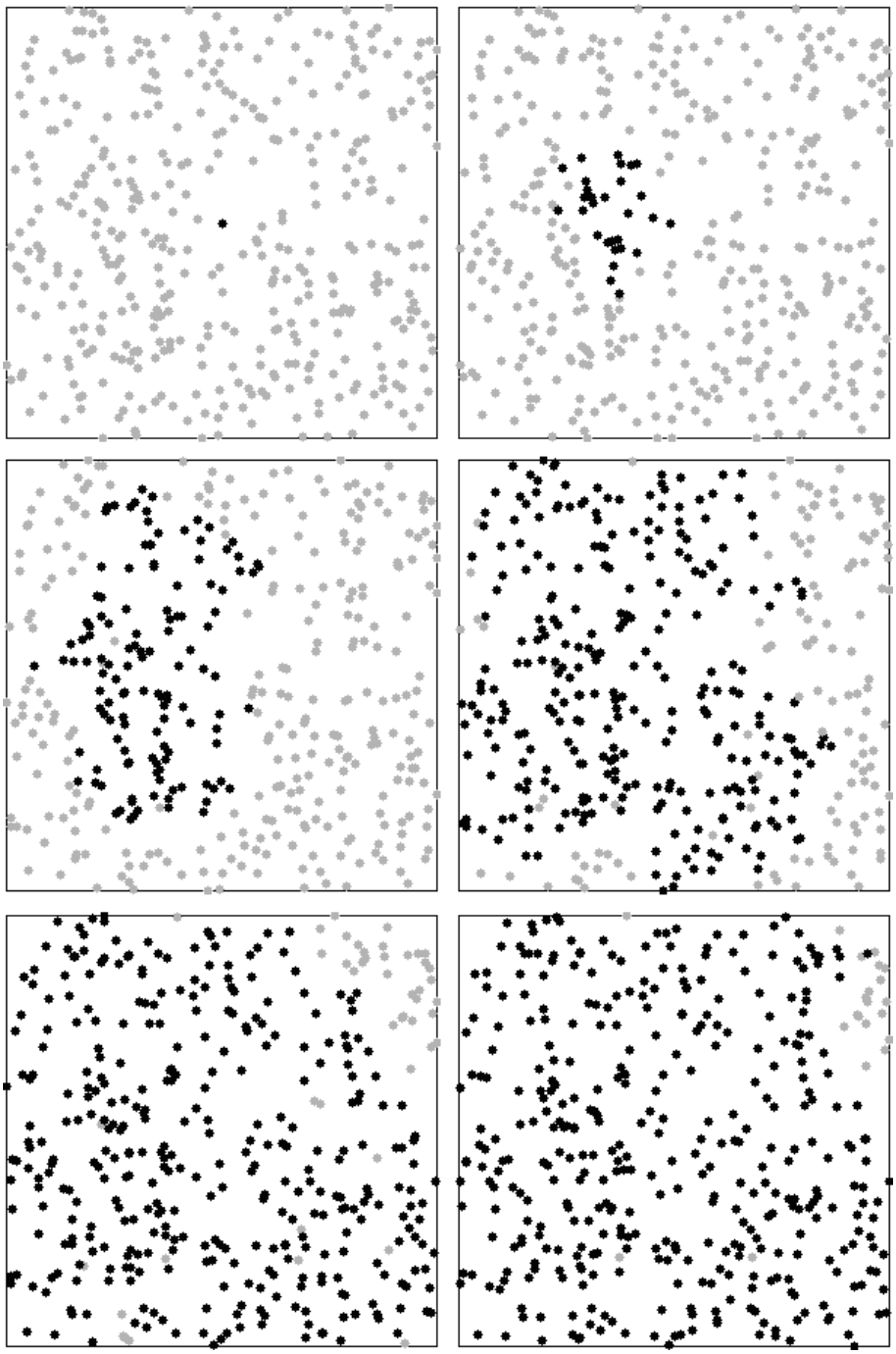


Figure 5.2: Broadcasting in a flying amorphous computer. The black dots represent the nodes that broadcast the message using protocol Broadcast2. The pictures from left to right and top to bottom represent subsequent time instants. The nodes are moving slowly, see the difference between the first and the last picture.

The running time of the broadcast protocol is determined by the node that starts the broadcast with parameter $l$. We assume that there will be a global constant $l$ that will be chosen at the time of the flying computer's assembly. We assume the value of that constant would initially be pre-configured into all nodes and all nodes would start all subsequent broadcasts with that value.

A reasonable value of $l$ depends on the number of nodes, their density and moving speed and also on the required reliability of the broadcasting operation. Since the message spreads through the amorphous computer with a limited speed, for small value of $l$ the message will be broadcast only to nodes at some distance from the starting node. Nodes further than this distance are out of reach for the message. If $l$ is increased, there is some value at which the range of reachable nodes will be larger than the size of the target area, i. e., all nodes will be in reach. However, even then there can be some nodes that do not receive the message, due to the stochastic nature of the broadcasting. Increasing the value $l$ still further allows more time for the broadcasting and therefore increases the probability that all nodes receive the message. In practise, the best value of $l$ must be chosen experimentally (cf. section 5.4).

The Broadcast2 protocol is not reliable in the sense that there is no bound on the error probability in message delivery. This is because we cannot analytically derive the probability that during the broadcasting time a path leading to a particular node would form.

When a message broadcast is started by a node, the result may be one of the following three:

- the message is received by all nodes,
- the message is received by some nodes,
- the message is not received by any node.

All algorithms running on the flying amorphous computer must work correctly if any of these cases happen. However, an algorithm may require that information is delivered to a particular node before the algorithm can proceed. We add the following condition to assure that such an algorithm does not get stuck.

**Definition 5.2** We call a flying amorphous computer *lively flying* if it satisfies the following condition: For each two nodes $X$ and $Y$, if $X$ is repeatedly sending message $m$ then after a finite (but unknown beforehand) number of broadcasts $Y$ receives message $m$.

If the flying amorphous computer is lively flying and a node keeps broadcasting the same message then it is guaranteed that at some time the message is successfully delivered to any other node. We use this feature in designing all algorithms for a flying amorphous computer.

The general pattern that we use to assure reliable delivery is the following. A node sends a message addressed to some other node or group of nodes using the broadcasting algorithm. Then all nodes wait until the broadcasting algorithm finishes. After that, the recipient sends an acknowledgement back to the sender. Then nodes again wait until the broadcasting algorithm finishes. If the originating node does not receive acknowledgement in time neccessary for two broadcasts, it assumes the sending was unsuccessful and repeats the process. This process is repeated possibly forever. However, if the amorphous computer is lively flying at some time the process will succeed and then the algorithm ends this loop.

## 5.2 Address Assignment

After we obtain fresh nodes from the factory, they are initially indistinguishable and their memory registers are zeroed. First we need to do a pre-configuration. Assume that we have already decided how many nodes we would use for our amorphous computer. We gather all the nodes in one place and using a sufficiently powerful radio we send a message to all nodes that contains the value of parameters $l$ and $p$ used for broadcasting (see algorithm 1). All nodes will use these parameters from now on. Then we spread the nodes to their target area.

After the amorphous computer is powered on, it starts flying. Then, we perform a set-up of the amorphous computer—we run an algorithm that assigns addresses to the nodes. The address assignment algorithm must be such that it runs correctly on a flying amorphous computer with the unreliable sending protocol described in the previous section (all Send instruction are performed using the Broadcast2 protocol). Especially, the algorithm must lead to a correct result even if some messages fail to be delivered to some nodes. The full algorithm is given as two parts—first part is the procedure carried out by the base node, presented on the next page as Algorithm 2 and Algorithm 3. The second part are the procedures (called message handlers) carried out by the nodes upon reception of various messages. The second part is presented as Algorithm 4 and Algorithm 5.

The assignment of addresses works according to a simple scenario described already in section 4.9.1. First, one of all nodes is selected and address number 1 is assigned to it. Then one node of the rest is selected and

address number 2 is assigned to it, and so on until all addresses are assigned. The selection of just one of all available nodes, so called leader election, is done by a series of splittings. Each splitting removes half of the candidates, in average. After a sufficient number of splitting we are left with only one candidate with high probability.

We now describe the algorithm in detail. The main procedure *InitAddresses* takes two arguments: *count* and $M$. Argument *count* gives the number of candidate splittings that are necessary so that only one of all nodes is selected with high probability. Argument $M$ gives the number of addresses that should be assigned. Initially, $M$ will be set to $N-1$, i.e. the number of all nodes except the base node, which will not have any address. The main procedure performs a loop in which procedure *SetNodeAddress* is called for each address.

During the address assignment, all nodes go through different internal states. The internal state is captured by the variables $phase1$, $phase2$ and $isSplit$. These internal variables are used to make sure that a node performs all necessary splittings in a series. If a node misses some messages because it is temporarily unreachable, there would be a discrepancy between the values of its internal variables and the values contained in the message. This way the node discovers that it missed some messages and will not react to any further messages in the sequence until a reset message makes it start again.

The procedure *SetNodeAddress* is divided into three parts. First is the initialization part. It consists of sending the message [New selection, $a$]. The message is being sent in a loop until at least one node replies, which means that it has accepted the message and executed the code linked to the message. The message handler for message [New selection, $a$] acts on all nodes that do not yet have any address assigned. Initially this is the case for all nodes (this can be seen in the message handlers, for the event of node start up). Upon reception of the message [New selection, $a$] the node updates its internal variables $phase1$, $phase2$ and $isSplit$. Only the nodes that have $phase2$ equal to $a$ are candidates to compete for the address $a$. The variable $phase1$ counts the number of symmetry breaking rounds (i.e. splittings) and all nodes must start with this variable set to 1.

The second part of the procedure consists of the symmetry breaking loop. The number of iterations of this loop is given by the argument *count*. In each round only the nodes with $phase1$ equal to the current round number take part in the competition. If the node wins in the competition, value of its $phase1$ variable is increased and the node passes to another round. The symmetry breaking round starts by the operation that splits the nodes into two groups A and B. This is done by the message [Split, $i$, $a$]. This message is sent in a loop so as to make sure that at least one node did the splitting.

```
InitAddresses(count, M)  /* Assigns M addresses to nodes */
begin
    for i=1 to M do
        SetNodeAddress(count, i);
    end
end

SetNodeAddress(count, a)  /* Assigns address a to a node */
begin
    repeat                        /* initializes node selection */
        Send [New selection, a];
        Wait for response;
    until response received;
    for i=1 to count do           /* performs count splittings */
        Split();
    end
    repeat              /* sets address a to the winning node */
        Send [Set address, count + 1, a];
        Wait for response;
    until response received;
end
```

**Algorithm 2**: Address assignment algorithm for the base node.

```
Split()        /* Splits nodes to two groups, selects one */
begin
    repeat
        Send [Split, i, a];
        Wait for response;
    until response received ;
    b=false;
    repeat
        Send [Check A, i, a];
        Wait for response;
        if response received then
            repeat
                Send [Choose A, i, a];
                Wait for response;
            until response received ;
            b=true;
        else
            Send [Check B, i, a];
            Wait for response;
            if response received then
                repeat
                    Send [Choose B, i, a];
                    Wait for response;
                until response received ;
                b=true;
            end
        end
    until b==true ;
end
```

**Algorithm 3**: Split procedure.

```
On node startup
begin
    address=0;
end

On message [New selection, a]
begin
    if address==0 then
        phase1=1;
        phase2=a;
        isSplit=false;
        Send [OK];
    end
end

On message [Split, i, a]
begin
    if phase1 == i and phase2 == a then
        randomly assign to group A or B;
        isSplit=true;
        Send [OK];
    end
end

On message [Check A, i, a]
begin
    if phase1 == i and phase2 == a and isSplit and assigned to
    group A then
        Send [OK];
    end
end

On message [Check B, i, a]
begin
    if phase1 == i and phase2 == a and isSplit and assigned to
    group B then
        Send [OK];
    end
end
```

**Algorithm 4**: Address assignment message handlers, part 1.

```
On message [Choose A, i, a]
begin
    if phase1 == i and phase2 == a and isSplit and assigned to
    group A then
        phase1=phase1 + 1;
        isSplit=false;
    end
    if phase1 == i + 1 and phase2 == a then
        Send [OK];
    end
end

On message [Choose B, i, a]
begin
    if phase1 == i and phase2 == a and isSplit and assigned to
    group B then
        phase1=phase1 + 1;
        isSplit=false;
    end
    if phase1 == i + 1 and phase2 == a then
        Send [OK];
    end
end

On message [Set address, i, a]
begin
    if phase1 == i and phase2 == a then
        address=a;
    end
    if address == a then
        Send [OK];
    end
end
```

**Algorithm 5**: Address assignment message handlers, part 2.

Then we must find which of the two groups A or B contains at least one node. We perform a check by sending the messages [Check A, $i$, $a$] or [Check B, $i$, $a$]. These messages are sent in a loop until we receive a positive answer from one of these groups. The nodes of these group are considered winners and they pass to the next round. The nodes are informed of this by the message [Choose A, $i$, $a$] or [Choose B, $i$, $a$]. The message is repeated in a loop so that at least one node passes to the next round.

After the necessary number of symmetry breaking rounds has been performed the last winning node is assigned the address $a$. This is done by the message [Set address, $count + 1$, $a$]. The message is repeated in a loop to make sure that the node receives the message.

The node that lost in the symmetry breaking competitions will stop with the $phase1$ equal to the last round it was in. On the next [New selection, $a$] message it will start competing for the new address $a$ again from the first round.

Now we verify that the described algorithm runs correctly on a lively, flying amorphous computer. The correctness is expressed by the following two theorems.

**Theorem 12** *Let a lively flying amorphous computer have at least one node to which no address has been yet assigned. Then procedure SetNodeAddress (count, a) will assign address a to at least one node of a lively flying amorphous computer.*

PROOF: We must prove that at least one node enters the symmetry breaking loop, wins in all the rounds and finally receives address $a$. Initially, there is at least one node without an address. This node can process the [New selection, $a$] message. The message is repeated until at least one reply is received. On a lively flying amorphous computer the reply will be received after a fixed number of repetitions. When the reply is received, we know there is at least one node entering the symmetry breaking loop. Message [Split, $i$, $a$] is then repeated until a response is received. Then we know that at least one node has performed the split. Then messages [Check A, $i$, $a$] or [Check B, $i$, $a$] are repeated until a reply is received. When a reply is received we have a group that contains at least one node. Then messages [Choose A, $i$, $a$] or [Choose B, $i$, $a$] are repeated in a loop until a reply is received. When the reply is received we know that at least one node has entered the next round of symmetry breaking. Similarly, we can verify that there is at least one node in each round that passes to the next round. Finally, the [Set address, $count + 1$, $a$] message is repeated in a loop until a reply comes from a node that passed through all the rounds. □

**Theorem 13** *Assume that no node of a flying amorphous computer has address $a$. Let the amorphous computer have $N$ nodes to which no address has been assigned. The procedure SetNodeAddress(count, a) with count $= \log_2((N-1)/\varepsilon)$ will assign address $a$ to more than one node with probability at most $\varepsilon$.*

PROOF: As we have seen in the proof of the previous theorem, address $a$ can be assigned to a node only after it has passed *count* symmetry breaking rounds. Let $X$ be one of the nodes that passes all rounds. On each round the nodes split into two groups. The probability that in all splittings some of the other $N-1$ nodes has entered the same group as $X$ is

$$\frac{N-1}{2^{count}} = \frac{(N-1)\varepsilon}{N-1} = \varepsilon.$$

□

In the case that two nodes get the same address, the simulation of RAM may not proceed correctly. We can make this probability arbitrarily small by increasing parameter *count*.

## 5.3 Simulation of a RAM

Let $M_1$ be a unit cost RAM with $N$ registers. A computation of $M_1$ can be simulated on a flying amorphous computer with at least $N+1$ nodes (one of them beeing the base node). We now show how the operation of the flying amorphous computer should be organised.

After the flying amorphous computer is assembled and turned on, we run the address assignment algorithm assigning addresses to all nodes. A node with address $a$ will simulate the contents of a register of $M_1$ with address $a$. The initial value is entered into each node through the base node. By using [Write, $a$, $x$] message the value $x$ is written into register with address $a$. This message is repeated in a loop until a reply from a node with address $a$ is received to confirm that the write was successfull. The filled in values are the input data for the computation. Then we may start to simulate the computation of $M_1$.

**Simulation algorithm:** The contents of the program counter is stored in the base node. The contents of the register $i$ is stored in the node with address $i$. Each operation may consist of reading some registers, writing to a register, computation, comparison and branching. The base node simulates the operation of the control unit of a RAM and also performs the computations, comparisons and branching. When the base node needs to read a value

from a register with address $a$ it sends message [Read, $a$]. The base node then waits for a reply. This is repeated in a loop until a reply is received. A node with address $a$ replies on the message Read with message [Value, $x$], where $x$ is the value stored in the node. If the base node must write value $x$ to register with address $a$, it sends message [Write, $a$, $x$]. The base node then waits for a reply. This is repeated in a loop, until a reply is received. Upon receiving the Write message the node with address $a$ updates its value to $x$ and sends reply [OK]. After an instruction is processed the base node updates the program counter and continues with the next instruction.

**Theorem 14** *Let $M_1$ be a RAM with $N$ registers of length $s_1$ bits. Let $M_2$ be a lively flying amorphous computer with $N+1$ nodes of size $s_2$ bits, $s1 \leq s2$, $N \leq 2^{s_2}$. Let the nodes of $M_2$ have assigned unique addresses and contain the input values. The simulation algorithm on a lively flying amorphous computer $M_2$ correctly simulates the computation of $M_1$ on a given input. The simulation ends in a finite time.*

Proof: In the proof we show that the following invariant holds: after each simulated instruction the contents of each node is the same as the contents of the corresponding register of $M_1$ would be. Initially, this is fulfilled thanks to the assumptions of the theorem. The computer $M_2$ has the same number of ordinary nodes as $M_1$ and the size of the registers $s_2$ is the same or larger than the simulated size $s_1$. So all input data can be fit into the memory of $M_2$.

Now we show that the invariant is not violated after simulation of any instruction. If the instruction requires a read of a register then this read must be simulated correctly, i.e. correct value must be read from the register. After the Read message is sent, thanks to unique address assignment, there is exactly one node that has the required register value and that can answer the message. The value in the node is correct according to the invariant. If the sending operation is unsuccessful, the reading is retried in a loop. So, the computation of $M_2$ holds on until the read operation is performed successfully. Because the amorphous computer is lively flying, the read message and its reply will be delivered successfully after a finite number of repetitions.

When a write operation is required, the Write message is sent. There is again exactly one register that can accept the new value and send a reply. So, if a reply is received by the base node, the write operation has been performed correctly. If no reply is received, the write operation retried in a loop and the computation is held on.

The computation of expressions is done by the base station and so it is always correctly performed. Similarly, the comparison, branching and updat-

ing of the next instruction register is always correct. We have also verified that the invariant holds, which also proves the correctness of one RAM instruction realisation. Each step of the computation requires delivering a fixed number of messages. On a lively flying amorphous computer the message will be successfully delivered in a finite number of steps so the whole computation ends in a finite time. □

## 5.4 Experimental validation of the communication protocol

We have shown how the flying amorphous computer simulates a RAM. We have also shown that on a lively flying amorphous computer the simulation is correct. However, we were unable to estimate the time complexity, i.e. the speed of the computation. That is because the working of the communication protocol depends on the topology of the communication network that we are not able to predict.

To get an insight into the situation, we can program a simulation of the amorphous computer. Then, by observing the behaviour of the simulation we can verify that in several randomly picked cases the communication protocol works as expected. We can also re-run a test several times starting from different initial positions to get some statistical description of the speed of communication in an amorphous computer in some concrete scenario.

The simulation can give us result only for cases with limited number of nodes. A model describing asymptotic behaviour of the communication protocol when increasing the number of nodes would be welcomed. However, so far we do not know how to develop such a model.

**What is tested**

We measure the average time needed for sending a message from the base node to another node and receiving the reply from that node by the base node. The time is measured for a range of different node speeds.

**Parameters**

In all experiments we used a fixed value $N$=400 for the number of nodes and the density $d$=6. We vary the node speed $v$ and the parameters $l$, $p$ of the Broadcast2 protocol. We measure the node speed $v$ relatively to $r$ and $T$. That is, a node with speed 1 travels distance $r$ in time $T$.

**Experiment set-up**

First we choose values of the parameters $v$, $l$ and $p$. Then, we generate random positions and random movement vectors for all nodes. We choose a random node $X$, different from the base node, to which the base node will send a message. Then the base node starts sending a message $m_1$. The simulation program simulates node movement and node communication. All nodes operate according to the Broadcast2 protocol. If message $m_1$ is received by node $X$ then $X$ sends a reply $m_2$ to the base node. The sending is successful if the base node receives $m_2$. If the sending is unsuccessful, the whole process is repeated till a successful sending occurs. After a sending was successful a new random positions and movement vectors are generated and the measurement continues with a new starting position. By this experiment we compute the average number of retries needed till a sending is successful. When we multiply this number by the time needed for one sending we get the average time needed for a successful sending.

**Results**

We performed tests for nodes moving with high speed, medium speed and low speed. We start by describing the medium speed.

We choose the speed 0.1 to be a reasonable medium speed for an amorphous computer. First, we observed a graphical animation of an amorphous computer with this speed to get some qualitative description of how the communication protocol behaves. We see that after a base node starts sending the message, the message is spread to surrounding nodes in waves of roughly circular shape. This is like in Model 2 of amorphous computer, which used static nodes. However, the message is also spatially spread by the movement of the nodes. The combination of both effects—message relaying between nodes and the nodes' movement—enables the message to spread faster. After the message is spread through nearly the entire amorphous computer, there may still be some individual nodes that did not receive the message. This happens for nodes that were most of the time surrounded by large number of neighbours and the message could not reach them due to permanent collisions.

Now, we present the quantitative results that we obtain by averaging over several tests. The results for speed 0.1 are given in figure 5.3 and in table 5.1. The values are the average time needed to send a message and to receive a reply. The time is measured in units $T$ (the time needed by a node to send a message). Each value is obtained from 600 tests. Since we do not know beforehand which parameters of the Broadcast2 protocol will yield the best results, we try several combinations to find the optimum value. From the results we see that for 400 nodes the best time is obtained for parameters

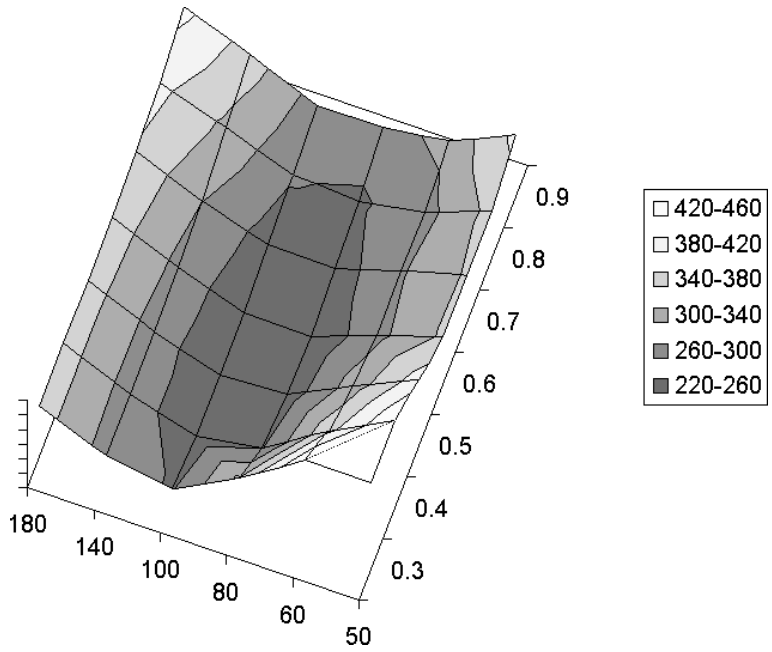



Figure 5.3: Average time needed to send a message and receive a reply. Parameter $l$ of Broadcast2 protocol ranges from 50 to 180, parameter $p$ ranges from 0.3 to 0.9. Darker color represents better values.

| $l$ | $p = 0.3$ | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
|---|---|---|---|---|---|---|---|
| 50 | 750 | 472 | 400 | 347 | 345 | 351 | 390 |
| 60 | 554 | 369 | 320 | 288 | 295 | 286 | 310 |
| 80 | 347 | 259 | 235 | 223 | 239 | 253 | 286 |
| 100 | 256 | 230 | 224 | 226 | 242 | 264 | 282 |
| 140 | 284 | 281 | 282 | 290 | 301 | 331 | 361 |
| 180 | 360 | 361 | 362 | 366 | 376 | 391 | 432 |

Table 5.1: Data for graph 5.3.

$l = 80$, $p = 0.6$, the time is 223 $T$. We see that the algorithm works well also for values of $p$ in the range from 0.4 to 0.8. This shows that in a real implementation it should not be difficult to find a well-performing value $p$. As we have seen in the proof of Theorem 4, low probability $p$ is good for nodes with high number of neighbours, while large $p$ is good for nodes with low number of neighbours. In a flying amorphous computer the number of neighbours is continually changing, which probably is the reason why the protocol works well for both low and high values of $p$.

Next, we consider the case of nodes moving with high speed. We choose speed 1 to represent a high speed. At this speed a node meets more neighbours during the time needed to send one message than in the medium speed scenario. This means that there are more nodes that can cause collisions. Also, there are less neighbours from which a message can be received. If two nodes move in different directions, they will move away from each other faster than the time required to send a message. So, message can be transferred only between nodes moving at similar directions. In the high speed case the nodes that received the message are located in a circular shape around the node on which the message originated, similarly as in the medium speed case. However, now the circular shape is only sparsely filled with nodes having the message, the rest are nodes that did not receive the message although they are near to the centre of the shape. The message quickly spreads through the whole area of an amorphous computer, and then it gradually spreads to other nodes uniformly in the whole area.

In the repeated experiments we find that for speed 1 the communication protocol performs best with values $l = 90$, $p = 0.3$. The average time is 265 $T$.

Next, we test the case of slowly moving nodes. We choose speeds 0.01, 0.001 and 0.0001 to represent low speed. With speed 0.01 the amorphous computer behaves very similarly to the medium speed case. However, to a significant degree the message is not transported by the node movement. Rather, it is only transported by the message relaying between nodes. This makes the running time of the protocol a bit longer. When we set the speed 0.0001, the amorphous computer starts to resemble the static Model 2 of amorphous computer. If the base node and the target node are in the same network component, then the message is successfully transmitted in a short time as in the case of speed 0.01. If they are in different components, then the amorphous computer must keep retrying the broadcasting till the two components become connected. This can take a long time. However, in the average time this large number of retries is averaged with the cases when the nodes are in the same component. As we have seen in Model 2, with density 6 most of the nodes are connected to the one largest network component.

So, the overall average time is not too high.

From the experiments we get that the best average time for $v = 0.01$ is 252 $T$, for $v = 0.001$ it is 315 $T$ and for $v = 0.0001$ it is 604 $T$.

The average times for different speeds are summarised in graph 5.4.

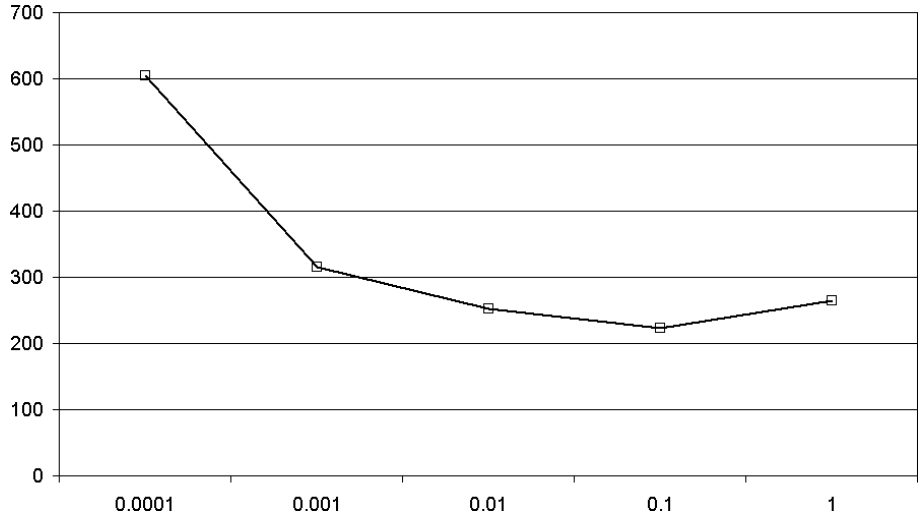



Figure 5.4: Average time needed to send a message and receive a reply. The speed $v$ goes from 0.0001 to 1.

**Conclusion**

We have verified that the proposed communication protocol works well over a broad range of node movement speeds. The factor between the highest and the lowest speed we tested was 10 000. We found that the protocol works best with speed 0.1, where there is a combined effect of message spreading by node to node message relaying and of the spatial message transport by the node movement.

# Chapter 6

# Conclusion

## 6.1 Achievements

We have analysed the concept of the amorphous computing from the computational complexity point of view. This work brought in several new results:

- Minimalistic model of an amorphous computer. We have defined a formal model of an amorphous computer that assumes minimal amount of computational resources in each node. Each node has radio transceiver that transmits on a shared channel and cannot detect collisions. Each node has memory of size $O(\log N)$, where $N$ is the number of nodes. All nodes start with the same internal data, i.e. having no addresses. Each node has random number generator. To analyse the problem on a finer scale, we have defined three special variants of the model that differ in the placement of the nodes—the first model used densely placed nodes, the second one used sparsely placed nodes and the third one used mobile nodes.

- Communication protocols. We have developed communication protocols that work on our models of amorphous computer. Due to the lack of any knowledge about position of individual nodes, the communication is done by broadcasting in all directions. Also, the nodes do not have initially any addresses and do not know their neighbours, so acknowledgements cannot be used to ensure reliable transmission of data. We achieve high reliability of transmission be repeating the same message for a sufficient number of times. The communication protocols provide minimal framework that later enables to assign unique addresses to nodes.

- The proof of universality. We have shown that infinite families of our minimal models have universal computing power. For the first model this was shown by simulating the computation of cellular automata. For the second and third model this was shown by simulating a RAM machine.

- A part of the results was presented in

  – L. Petru. On the Computational Power of an Amorphous Computer. WDS'04 Proceedings of Contributed Papers, Prague, CZ, pp. 156–162, June 2004

  – L. Petrů. Amorfní počítání: model univerzálního počítače sestaveného z jednoduchých kooperujících agentů. Proceedings of Kognice a umělý život VI, Opava, CZ. pp. 309–314, May 2006

  – L. Petrů, J. Wiedermann. A Model of an Amorphous Computer and Its Communication Protocol. SOFSEM 2007: Theory and Practice of Computer Science, LNCS Volume 4362, pp. 446–455, July 2007

  – J. Wiedermann, L. Petru. Computability in Amorphous Structures. Computation and Logic in the Real World, LNCS Volume 4497, pp. 781–790, July 2007

  – J. Wiedermann, L. Petru. Communicating Mobile Nano-Machines and Their Computational Power. Proceedings of Nano-Net 2008, 3rd International Conference on Nano-Networks, Boston, USA, Sept. 2008

  – J. Wiedermann, L. Petrů. On the Universal Computing Power of Amorphous Computing Systems. Theory of Computing Systems, Volume 45, Number 4 / November 2009, pp. 995–1010, Springer New York, Jan. 2009

  – L. Petrů, J. Wiedermann. Létající amorfní počítač. Proceedings of Kognice a umělý život IX, Opava, CZ, pp. 263–266, May 2009

  – J. Wiedermann, L. Petrů. Communicating Mobile Nano-Machines and Their Computational Power (Extended abstract), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Nano-Net Third International ICST Conference, NanoNet 2008, Boston, MA, USA, Revised Selected Papers, pp. 123–130, July 2009

Our results have applications also in other fields of computer science where systems consisting of large number of simple parts are used, e.g. specifically for sensor networks. Each sensor network would fulfill the requirements

of our amorphous computer model, and so each sensor network can operate as a universal computer. Given that, a sensor network is not only able to observe data of its environment but also perform some processing of the data, without needing any external computer. This might be of value for sensor networks that are dispersed in a very remote area where direct communication with external operator is problematic to achieve. This could be the case for example when exploring other planets of the solar system.

## 6.2 Further research

There are several open directions in which the research can be extended.

First, it would be interesting to find out if universal computation can be achieved for the model using sparsely placed nodes with asymptotically less than $\sim \log N$ internal memory, or to prove that it is not possible. An initial step in this way has been done in our article on communicating nanomachines [41]. By simulating a counter machine, it was possible to use nodes with constant memory size, there. However, the nodes required an extra timer with adjustable tick time (so-called probabilistic timed automata), and the maximum value of this tick time depends on the number of nodes $N$.

Another interesting direction would be to find a model for message spreading on our model of flying amorphous computer. In the static case, the model of our network corresponded to the continuum percolation model. In the case of the flying amorphous computer model a link to dynamic percolation theory ([21], [29]) could exist. The dynamic percolation theory is used, e. g., to describe the transport of electric charge in polymer solid electrolytes.

Another direction would be to try to apply our results to the field of synthetic biology. In synthetic biology, the researchers are trying to identify some standard parts of living systems and engineer novel organisms by putting together these parts in a completely new way (see [16], [17]). It is even possible to simulate some simple gates like AND and OR, and it is possible to perform cell to cell communication using specific molecules. Given a mathematical model of communication of these synthetic cells and given that the cells could be engineered supporting certain minimum computational capabilities, a group of such cells could operate as an amorphous computer. This would have applications, e. g., for new intelligent drugs.

# Bibliography


[1] H. Abelson, et al. Amorphous Computing. MIT Artificial Intelligence Laboratory Memo No. 1665, Aug. 1999

[2] H. Abelson, D. Allen, D. Coore, Ch. Hanson, G. Homsy, T. F. Knight, Jr., R. Nagpal, E. Rauch, G. J. Sussman, R. Weiss. Amorphous Computing. Communications of the ACM, Volume 43, No. 5, pp. 74–82, May 2000

[3] H. Abelson, J. Beal, G. J. Sussman. Amorphous Computing. Computer Science and Artificial Intelligence Laboratory, Technical Report, MIT-CSAIL-TR-2007-030, June 2007

[4] A. V. Aho, J. E. Hopcroft, J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974

[5] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. A Survey on Sensor Networks. Communications Magazine, IEEE, Vol. 40, No. 8., pp. 102–114, 2002

[6] I. F. Akyildiz, T. Melodia, K. R. Chowdhury. A survey on wireless multimedia sensor networks. Computer Networks, Vol. 51, No. 4, pp. 921–960, March 2007

[7] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, R. Peralta. Urn automata. Technical Report YALEU/DCS/TR-1280, Yale University Department of Computer Science, 2003

[8] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, R. Peralta. Computation in networks of passively mobile finite-state sensors. Distributed Computing, Volume 18, Number 4, pp. 235–253, March 2006

[9] D. Angluin, J. Aspnes, D. Eisenstat. Fast Computation by Population Protocols with a Leader. Distributed Computing, LNCS Volume 4167, pp. 61–75, Sep. 2006

[10] D. Angluin, J. Aspnes, D. Eisenstat, E. Ruppert. The computational power of population protocols. Distributed Computing, Volume 20, Number 4, pp. 279–304, Nov. 2007

[11] D. Angluin, J. Aspnes, M. J. Fischer, H. Jiang. Self-stabilizing population protocols. ACM Transactions on Autonomous and Adaptive Systems, Volume 3, Issue 4, pp. 1–28, Nov. 2008

[12] D. K. Arvind, K. J. Wong. Speckled Computing: Disruptive Technology for Networked Information Appliances. In Proceedings of the IEEE International Symposium on Consumer Electronics (ISCE'04) (UK), pp 219–223, Sep. 2004

[13] R. Bar-Yehuda, O. Goldreich, A. Itai. On the Time-Complexity of Broadcast in Multi-hop Radio Networks: An Exponential Gap Between Determinism and Randomization. J. Comput. Syst. Sci. Vol. 45, No. 1, pp. 104–126, 1992

[14] J. Beal, J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. IEEE Intelligent Systems, Volume 21, Issue 2, pp.10–19, April 2006

[15] J. Beal. Programming an Amorphous Computational Medium. Unconventional Programming Paradigms. LNCS Volume 3566, pp. 121–136, 2005

[16] Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, E. Shapiro. An autonomous molecular computer for logical control of gene expression. Letters to Nature, Nature 429, pp. 423–429, May 2004

[17] S. A. Benner, A. M. Sismour. Synthetic Biology. Nature reviews. Genetics. Volume 6, pp. 533–543, July 2005

[18] W. Butera. Programming a paintable computer. PhD Thesis, Massachusetts Institute of Technology, Cambridge MA, 2001

[19] D. Coore. Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer. PhD thesis, Massachusetts Institute of Technology, Cambridge MA, 1999

[20] J. Dall, M. Christensen. Random geometric graphs. Phys. Rev. E 66, 016121, July 2002

[21] S. D. Druger, M. A. Ratner, A. Nitzan. Generalized hopping model for frequency-dependent transport in a dynamically disordered medium, with applications to polymer solid electrolytes. Phys. Rev. B, Volume 31, Issue 6, pp. 3939–3947, 1985

[22] M. Franceschetti, J. Bruck, L. J. Schulman. A random walk model of wave propagation. IEEE Transactions on Antennas and Propagation, Volume 52, Issue 5, pp. 1304–1317, May 2004

[23] I. Glauche, W. Krause, R. Sollacher, M. Greiner. Continuum percolation of wireless ad hoc communication networks. Physica A: Statistical Mechanics and its Applications, Volume 325, Issues 3–4, pp. 577–600, July 2003

[24] G. Grimmett. Percolation. Springer, 1999

[25] J. M. Kahn, R. H. Katz, K. S. J. Pister. Next century challenges: mobile networking for "Smart Dust". Proc. of the 5th annual ACM/IEEE international conference on Mobile computing and networking, Seattle, Washington, United States, pp. 271–278, 1999

[26] J. M. Kahn, R. H. Katz, K. S. J. Pister. Emerging Challenges: Mobile Networking for "Smart Dust". Journal of communication and networks, vol. 2, No. 3, pp. 188–196, Sep. 2000

[27] R. Nagpal. Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. PhD Thesis, Massachusetts Institute of Technology, 2001

[28] R. Nagpal. Organizing a global coordinate system from local information on an amorphous computer. Technical Report AI Memo 1666, MIT, 1999

[29] A. Nitzan, M. A. Ratner. Conduction in polymers: Dynamic disorder transport. Journal of Physical Chemistry, Issue: 98:7, pp. 1765–1775, 1994

[30] L. Petru. On the Computational Power of an Amorphous Computer. WDS'04 Proceedings of Contributed Papers, Prague, CZ, pp. 156–162, June 2004

[31] L. Petrů. Amorfní počítání: model univerzálního počítače sestaveného z jednoduchých kooperujících agentů. Proceedings of Kognice a umělý život VI, Opava, CZ. pp. 309–314, May 2006

[32] L. Petrů, J. Wiedermann. A Model of an Amorphous Computer and Its Communication Protocol. SOFSEM 2007: Theory and Practice of Computer Science, LNCS Volume 4362, pp. 446–455, July 2007

[33] L. Petrů, J. Wiedermann. Létající amorfní počítač. Proceedings of Kognice a umělý život IX, Opava, CZ, pp. 263–266, May 2009

[34] V. Vinge. A Deepness in the Sky. Tor Books, 800 pages, Jan. 2000

[35] B. Warneke, B. Atwood, K. S. J. Pister. Smart dust mote forerunners. The 14th IEEE International Conference on Micro Electro Mechanical Systems, MEMS 2001, pp. 357–360, Jan. 2001

[36] B. Warneke, M. Last, B. Liebowitz, K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. Computer, Volume 34, Issue 1, pp. 44–51, Jan. 2001

[37] B. A. Warneke, K. S. J. Pister. An ultra-low energy microcontroller for Smart Dust wireless sensor networks, IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC. Vol. 1, pp. 316-317, Feb. 2004

[38] R. Weiss, S. Basu, S. Hooshangi, A. Kalmbach, D. Karig, R. Mehreja, I. Netravali. Genetic circuit building blocks for cellular computation, communications, and signal processing. Natural Computing: an international journal, Volume 2, Issue 1, pp. 47–84, April 2003

[39] J. Wiedermann, L. Petru. Computability in Amorphous Structures. Computation and Logic in the Real World, LNCS Volume 4497, pp. 781–790, July 2007

[40] J. Wiedermann, L. Petrů. On the Universal Computing Power of Amorphous Computing Systems. Theory of Computing Systems, Volume 45, Number 4/November 2009, pp. 995–1010, Springer New York, Jan. 2009

[41] J. Wiedermann, L. Petru. Communicating Mobile Nano-Machines and Their Computational Power. Proceedings of Nano-Net 2008, 3rd International Conference on Nano-Networks, Boston, USA, Sept. 2008

[42] J. Wiedermann, L. Petrů. Communicating Mobile Nano-Machines and Their Computational Power (Extended abstract), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Nano-Net Third International ICST Conference, NanoNet 2008, Boston, MA, USA, Revised Selected Papers, pp. 123–130, July 2009

[43] Stephen Wolfram. A New Kind of Science. Wolfram Media, 2002, ISBN 1579550088

[44] http://www.dustnetworks.com/

[45] http://www.specknet.org/