FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Validation and Parallel Implementation of Spatial Agent Contacts in an Agent Based Simulation of Epidemics on Hexagonal Grids

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Pavol Bauer
Matrikelnummer 0825628

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dipl.-Ing.Dr.Techn. Felix Breitenecker
Mitwirkung: DI Florian Miksch

Wien, 01.09.2011

(Unterschrift Pavol Bauer) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Validation and Parallel Implementation of Spatial Agent Contacts in an Agent Based Simulation of Epidemics on Hexagonal Grids

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Pavol Bauer
Registration Number 0825628

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Dipl.-Ing.Dr.Techn. Felix Breitenecker
Assistance: DI Florian Miksch

Vienna, 01.09.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Pavol Bauer
Murlingengasse 15/3/344 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Pavol Bauer)

Acknowledgements

Mein größter Dank gilt meiner Frau Sanja, für hunderte Stunden an Geduld und die Unterstützung, ohne welche diese Arbeit nie vollendet worden wäre.

Weiters bedanke ich mich bei Prof. Breitenecker für die Betreuung und im Besonderem Florian Miksch für die detaillierten Korrekturen und eine kontinuierliche Verbesserung des vorliegenden Werkes.

Zu guter Letzt ein großer Dank an Toni Grünberg für die Bereitstellung der benötigten Ressourcen und eine Menge motivierender Worte während der Fertigstellung der Arbeit.

Danke euch allen!

Abstract

Mathematical modeling and simulation are important tools in the analysis of epidemics and may support decision makers to control and eventually avoid the spread of infections in the future. This work presents an epidemic model that creates an artificial population, so called “agents“, which are exposed to a virtual pathogen. Consequently, agents can be infected, transfer infection among other agents and eventually recover from the illness. This behavior relies on a so-called contact finding module that assigns agents to each other and generates possible epidemic contacts. The main focus of this thesis is to determine if this task may be simulated with either statistical methods or the help of an abstract space, which provides agents with movement and the possibility to contact others in the immediate vicinity. The structure of the space and the movement rules are inspired by the Lattice Gas Cellular Automaton (LGCA); a discrete mathematic method to simulate natural propagation as it appears in uncompressed fluids and gases. As simulations of large human populations using LGCA have proven to require high computation power, this work shows a developed parallel algorithm that can be executed on GPU, the main processor of the graphic card. The results show that this porting allows simulations of large spatial grids to be executed up to 100 times faster than those on CPU. In summary, this work analyzes the possibilities to simulate contact finding at an epidemic simulation and presents a parallel implementation, which can be used to achieve higher time and performance efficiency of future epidemic simulations.

Kurzfassung

Mathematische Modellierung und Simulation sind wichtige Hilfsmittel in der Analyse von Epidemieausbreitungen, welche Entscheidungsträgern die Möglichkeit bieten zukünftige Infektionswellen schneller erkennen und vorbeugen zu können. Im Rahmen dieser Diplomarbeit wird ein Epidemiemodell vorgestellt welches eine synthetische Bevölkerung - sogenannte “Agenten” - kreiert, welche einem virtuellen Krankheitserreger ausgesetzt werden. Die Agenten können mit diesem Erreger angesteckt werden, die Infektion auf andere Agenten übertragen und nach einer gewissen Zeitspanne wieder genesen. Die Basis für dieses Verhalten stellt ein Teil des Models dar, welches Agenten einander zuweist und einem möglichen Kontakt berechnet. Dieses Modul stellt den Hauptaugenmerk der Arbeit dar: es wird untersucht ob die Kontaktfindung mit statistischen Methoden oder mit Hilfe eines abstrakten Raumes auf welchem sich die Agenten frei bewegen und treffen können, simuliert werden kann. Die Struktur des Raumes und die Bewegung der Agenten sind an den Lattice Gas Cellular Automaton (LGCA) angelehnt, einer diskreten mathematischen Methode zur Simulation von natürlicher Bewegung wie sie in unkomprimierten Gasen oder Flüssigkeiten vorkommt. Da die Simulation einer grossen Bevölkerung mit Hilfe des LGCA sehr rechenaufwendig ist, ist im Rahmen der Diplomarbeit ebenfalls ein paralleler Algorithmus entwickelt worden welcher auf der GPU, dem Hauptprozessor der Grafikkarte, ausgeführt werden kann. Es hat sich gezeigt, dass diese Rechenarchitektur die vorgestellte Aufgabe bis zu 100-mal schneller als die CPU verarbeiten kann. Während die Analyse der Kontaktfindung mögliche Anwendungsfälle der Methoden beleuchtet stellt die parallele Implementierung eine Möglichkeit dar, zukünftige räumlich basierte Epidemiesimulationen preisgünstig und zeiteffizient auf gewöhnlichen Grafikkarten ausführen zu können.

Contents

CHAPTER 1

Introduction

1.1 Introduction to Mathematical Modeling

Numerous theoretical and scientific studies use *models* to solve problems whose solution is limited by an empirical approach. These limitations occur in form of experimental inaccessibility due to a high number of variables needed to be observed in real time, physical, technological or ethical restrictions; as in the case of epidemic models. One of the definitions says that “A mathematical model is an abstract, simplified, mathematical construct related to a part of reality and created for a particular purpose" [1]. Generally spoken, people working with models are aiming to understand, develop and optimize “systems". By that “system" refers to an object of interest that can be a part of the nature or an artificial system. Most of the natural and technological systems have a high complexity, so the general strategy used to deal with this complexity is “simplification". The definition supports the idea that the best model is the simplest model, that still serves its purpose of solution finding (Figure 1.1).

Most of the systems in our environment are example of continuously changing - *dynamic* systems. According to Breitenecker et al [3], there are two approaches to a dynamic system analysis. The first one is called deductive analysis and stands for theoretical modeling and the second one is named inductive synthesis and involves experimental modeling and identification. Theoretical modeling relies on physical, chemical and biological rules or social observations and consists of state variables presenting real system parameters. Unlike to this modeling approach, experimental modeling is mostly based on measurements data which are used in statistical models approaching a system to be described as a "black box", where parameters obviously do not represent real system parameters.

There are many ways to classify a mathematical model, some of the most common distinction between them can be given in following list:

- *Analytical or numerical* - A model can be either represented by an equation that can be solved analytically or by a set of mathematical instructions (e.g. differential equations) which can be computed with the help of a computer (e.g. ODE solver)

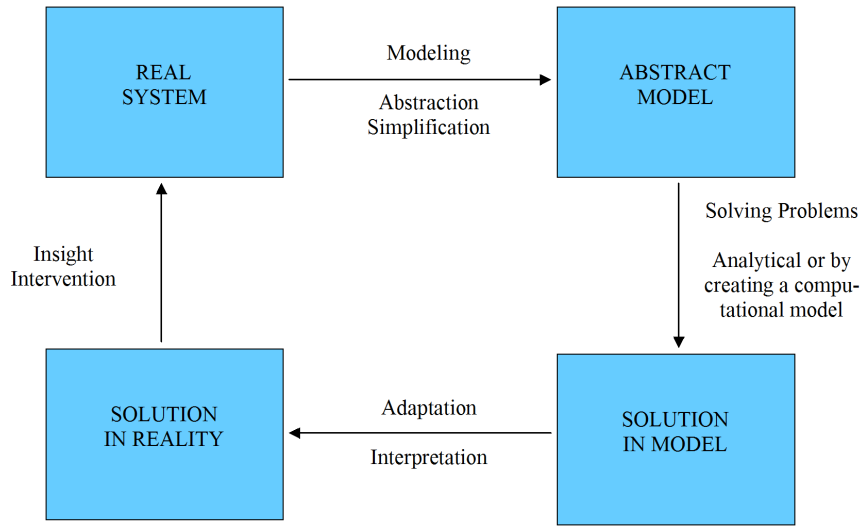

Figure 1.1: Finding problem solutions with models, adapted from [3]

- *Deterministic or probabilistic* - Transfer functions between model states are uniquely defined by parameters and by sets of previous states in a deterministic model. In a stochastic model on the other hand, transfers between states are not described by deterministic values but by probability distributions which provide the model with a level of randomness.

- *Discrete or continuous* - Discrete models are defined by a discontinuity of their time domain, resulting in a division to time steps - the smallest units of time discretization. Continuous models are not discretized and can be solved for any continuous time value (e..g solution of a different equation).

- *Black box or White box modeling* - According to how much a priori information of a system is available, models can be classified to black box or white box models. While no information is available about model structure or parameters in the black box model, the white box model offers a completely transparent overview of model structure, parameters and solving procedures. Practically, the most models are situated between the black box and white box paradigm [2].

Summarized, seven steps have been proven to be of a great importance for a reliable modeling study, known as a process flow called *simulation cycle*. As it is shown at Figure 1.2, as a first step a clear definition of a problem to be solved or question to be answered is essential, a procedure which can be entitled as *modeling*. In the next step, an accurate systems analysis is needed. This should include identification of parts of a system that are most relevant for the solution of a problem. This step also includes the development of a model based on the results

of the systems analysis which implies the choice of a reasonable modeling technique and the *implementation* of a model.

After a first model has been created, a first *basic simulation run* can be calculated. If the model is a numerical model, this task is carried out by a simulator. A simulator is a computer program or dedicated device which is able to numerically solve a formalized model with help of one or more numerical methods and a set of model and simulation parameters [2].

After the simulation completed the possibly most important steps in the simulation cycle has to be taking into account; the *validation* and *verification* procedures. The *validation* task ensures that a model meets the indeed requirements in terms of the methods employed and the results obtained. The goal of model validation is to make the model useful in the sense that the model addresses the right problem, provides accurate information about the system being modeled and reproduces the desired behaviors correctly. The *verification* ensures that the model is programmed correctly and does not contain errors, oversights or bugs. Moreover, it ensures that the specification is complete and the model performs as intended [4].

If the validation assurance fails (indicated by the lower green square at Figure 1.2) the process flow is directed to the verification control. If the verification declares insufficient quality of the model structure, the model has to be formulated again to find basic errors and possible wrong assumptions. If the model structure proves to be admissible, the parameters of the model have to be identified and the modeler can proceed with the analysis and implementation of the model before running the next simulation. Practically, the simulation cycle has to be processed many times before the right model structure, parameters and implementation have been found and the verification as well as validation tasks prove the model to be valid.

1.2 Simulation of Epidemics

The spread of infectious diseases is one of the major problems of nowadays global health. The definition of infectious diseases summarizes types of illness resulting from infections of pathogenic biological organisms such as bacteria, viruses, fungis or parasites. These infectious pathologies are also often described as communicable diseases or transmissible diseases due to their potential of transmission between human beings [5]. This transmission may occur through several different mechanisms. Respiratory diseases are commonly acquired by contact with aerosolized droplets (e.g. as produced by sneezing or coughing), whereas gastrointestinal diseases are often acquired by ingesting contaminated food or water. Sexually transmitted diseases are acquired through contact with body fluids, generally as a result of sexual activity. [5] The World health organization published a list of infectious diseases [6] that caused the most human deaths in the year 2002, presented in table 1.1.

It is well known, that some of these infectious diseases are mostly spread in developing countries of the African or Asian continent. They are often caused by inadequate hygienic or medical standards, as well as deficit of public health precautions. Nevertheless, there can be given examples of infectious diseases which have as well been a serious threat to industrial countries in Europe and North America. A historic example is the Spanish flu, which evolved in the United States and expanded to European countries in the beginning of the 20th century. It caused 50 to 100 million deaths, about 3% of the worlds population at the time [7]. Nowadays,

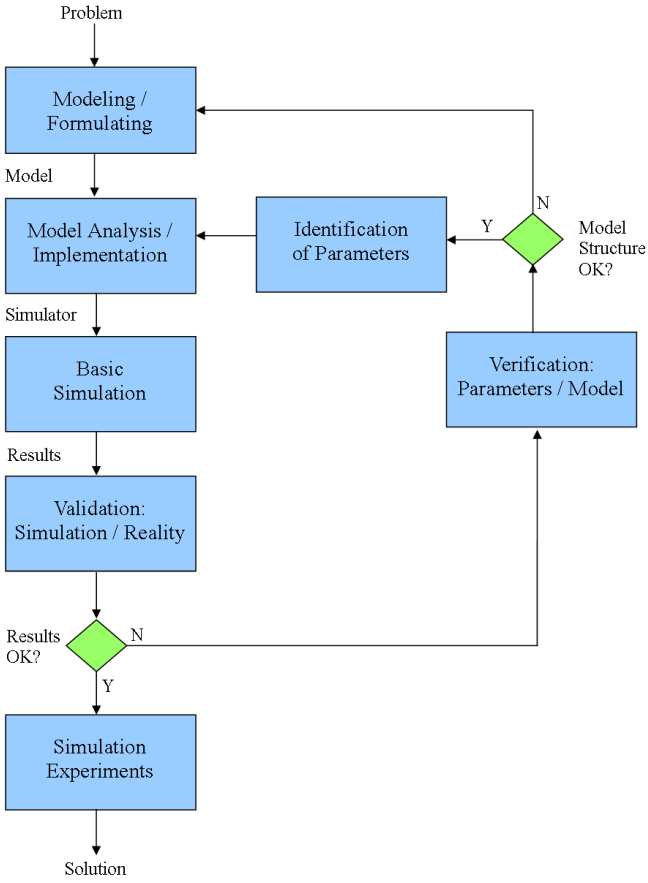

Figure 1.2: The simulation cycle [3]

Rank	Disease	Percentage of total deaths
1	Lower respiratory infections	6,9 %
2	HIV/AIDS	4,9%
3	Diarrheal diseases	3.2%
4	Tuberculosis	2,7 %
5	Malaria	2.2%
6	Measless	1.1%
7	Pertusis	0.5%
8	Tetanus	0.4%
9	Memingitis	0.3%
10	Syphilis	0.3 %
	All infectious diseases	25.9%

Table 1.1: Worldwide mortality due to infectious disease in 2002 [6]

different Influenza mutations are still dangerous, as it could be experienced while the worldwide spread of Influenza mutations H1N1 or H5N1 in the the years 2009 and 2010. As it can be seen in table 1.1, infectious diseases were the main cause of 26 % of worldwide deaths in the year 2002 (in total number this equals 14.7 million deaths) . This fact makes it necessary to control future epidemic spreads with the help of surveillance methods and continuous analysis of existent diseases as well as potential outbreaks. To analyze the future development of spreads mathematical modeling can be applied to estimate the endangerment of an infectious disease in respect to the transmission rate and speed and consideration of social and spatial factors. These methods may help to estimate a danger resulting from the infectious spread and carry out necessary precautions in time. In the next chapters a brief introduction on epidemic models and their use in combination with other simulation methods will be presented.

CHAPTER 2

Used Methods and Concepts

2.1 The SIR Model

One of the most popular models of infectious diseases is the so called SIR model, which has been first defined by W.O.Kermack in 1927 [14]. It simplifies the view on epidemics by dividing an assumed population in three groups:

- **Susceptible:** The group of healthy individuals, which are susceptible to the disease and can become infected contingently.
- **Infected:** The group of infected individuals, that are capable to spread the disease and infect population from the susceptible category. They may become recovered by a defined probability or a time period, dependent on the realisation of the model.
- **Recovered or Resistant:** Describes the group of individuals, that get recovered after the infection and then become healthy again. In this state, they are not able to infect again or transmit the infection to others.

The SIR Model can be considered as an abstraction of the basic epidemic behavior that broadly describes a group of common infection spreads. The exact model can be realized by different modeling techniques as well as top-down or bottom-up approaches. Common top-down implementations include compartment models or system dynamics. However, the most popular representation is defined by ordinary differential equations (ODEs):

$$\frac{dS}{dT} = -\beta SI \tag{2.1}$$

$$\frac{dI}{dT} = \beta SI - \mu I \tag{2.2}$$

$$\frac{dR}{dT} = \mu I \tag{2.3}$$

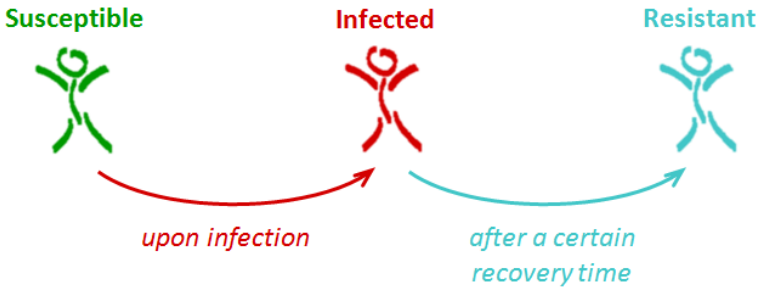

Figure 2.1: Schematic representation of the SIR model by W.O. Kermack

Where dS, dI and dR are the numbers of susceptible, infected and recovered individuals at a given time t. The parameter β denotes the growth of the infected group, it depends on the interaction between susceptible and infected individuals.

The basic reproduction ratio R_0 describes the expected number of new infections from a single infected individual in an susceptible population [14]. It is defined by:

$$R_0 = \frac{\beta}{\mu} \tag{2.4}$$

The model follows several assumptions. For example, the population size have to be constant over the observed period of time, so no birth and dead occur in the basic model. Additionally, the incubation time of the agents is zero and the length of infectivity is the same as the length of the disease [14].

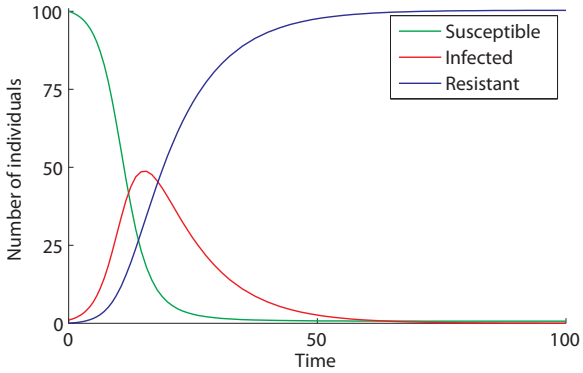

Figure 2.2: Solution of the ODE realization of Kermack's SIR Model, computed continuously on a starting population of 100 susceptible and 1 infected individual (Top-Down)

The solution of the equations show a typical trend, as can be seen in figures 2.2 and 2.3, that represent computation results of the ODE and Agent-Based SIR model. In comparison,

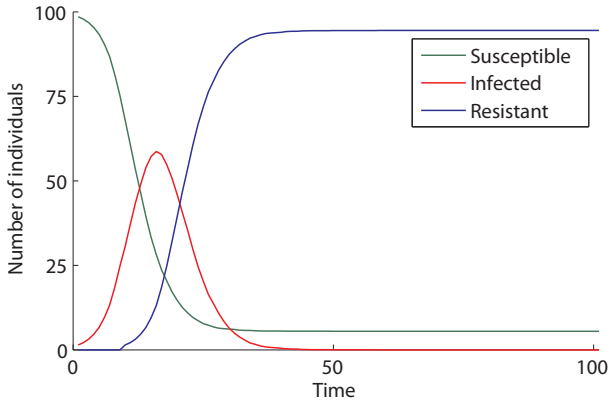

Figure 2.3: Solution of an agent-based simulation of the SIR Model, computed at discrete time steps and spatial contact generation on a rectangular grid (Bottom-Up)

oscillations can be observed in every group of the agent based model, due to the statistical deviation in the generation of potential contact between the individuals. In contrast to the top-down approach of the ODE based representation, a bottom-up implementation using Agend-Based modeling calculates SIR infection dynamics upon every single individual defined in the simulation. As it will be explained in detailed in the following chapters, the autonomous interaction of these agents at the “bottom" of the model forms the trend of a global result which can be equal to a directly calculated result of a top-down calculation.

There are many modifications of the basic SIR model, where the the transition flow $S \rightarrow I \rightarrow R$ can be exchanged to $S \rightarrow I \rightarrow R \rightarrow S$, to simulate a population that can get susceptible again after infection (assuming there is no immunity for the illness) or the introduction of a new group as the group of exposed individuals (denoted by the letter E), infected but not yet infectious to others, resulting in $S \rightarrow E \rightarrow I \rightarrow R$.

2.2 Agent Based Simulation

Agent-based modeling and simulation (ABMS) is an approach to modeling complex systems of interacting, autonomous “agents". An agent is an abstract entity, which denotes an independent component of a bigger system [12], a concept often used as an abstraction of human beings, taken from a population. Although there is no universal agreement on the precise definition of the term *agent* definitions tend to agree on some specific points. An agent must have a behaviour, which can range from primitive reactive decision rules to complex adaptive artificial intelligence (AI) [8] and an interaction with other agents. This makes it possible to agents model individually, so the full effects of the diversity that exists among agents in their attributes and behaviours can be observed and has an impact on the behaviour of the whole system [11]. This is the reason, why agent based modeling is often used to model social networks, self-organization, or as it will be presented in future chapters, individual interaction of subjects in a system. Because of

this, it belongs to the group of bottom-up modeling techniques, which are in contrast to top-down methods like system dynamics. Patterns, structures, and behaviours emerge that were not explicitly programmed, but arise through the agents behaviours and interactions [8].

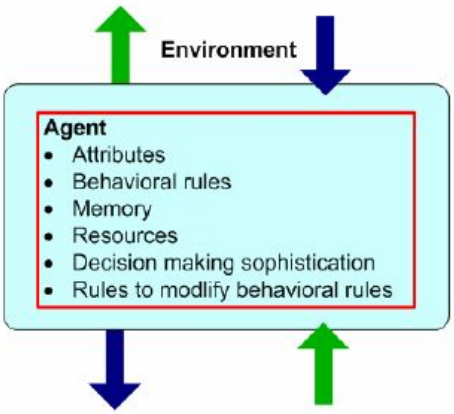

Figure 2.4: Description of an agent, as shown in [12]

As presented in [11] and [12] agents are considered to have following characteristics:

- An agent is a self-contained, modular, and uniquely identifiable individual. Under *modularity* we understand the fact that an agent has a boundary.
- An agent is *autonomous* and self-directed.
- An agent has a *state* that varies over time. As such, the richer the set of an agents possible states, the richer the set of behaviours that an agent can have. In an agent-based simulation, the state at any time is all the information needed to move the system from that point forward.
- An agent is *social*, having dynamic interactions with other agents, that influence its behavior.

Structure of an Agent Based Model

The reason, why agent-based modeling is becoming so widespread is the fact that we live in an increasingly complex world. The systems that we need to analyze and model are becoming more complex in terms of their dependencies [11]. Traditional modeling techniques as differential equations are much more difficult to adapt onto this complexity [12]. On the other hand there must be mentioned, that agent-based modeling is a computational intense method, when computational workload increases with the number of interacting agents in the system. Solving of problems with a high number of agents with numerous attributes and behaviours could not have been possible with consumer hardware in the last decades.

As Macal and North gave in their publication [11], a typical agent-based model has three elements:

- A set of agents, their attributes and behaviors.
- A set of agent relationships and methods of interaction: An underlying topology of connectedness defines how and with whom agents interact.
- The agents environment: Agents interact with their environment in addition to other agents.

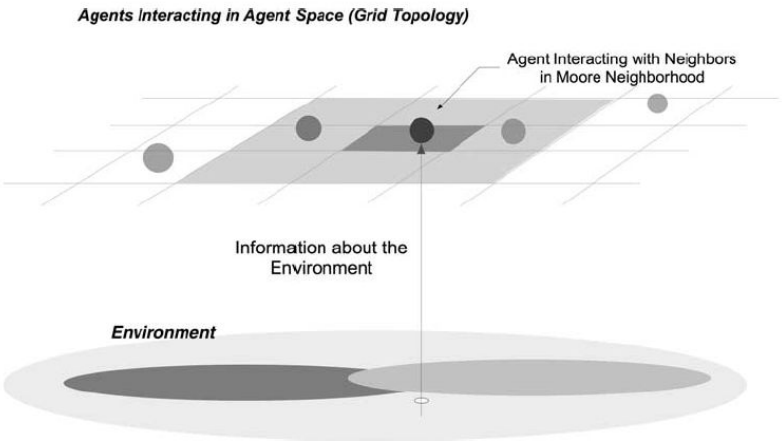

Figure 2.5: The structure of a typical agent-based model [11]

A possible structure of an agent-based model is shown in Figure 2.5. To run an agent-based model, agents have to repeatedly execute their behaviours and interactions. This process often does, but is not necessarily visualized over a timeline, as in time-stepped, activity-based, or discrete-event simulation structures [11].

Agent-based modeling concerns itself with modeling agent relationships and interactions as much as it does modeling agent behaviors. The two primary issues of modeling agent interactions are specifying who is, or could be, connected to whom, and the mechanisms of the dynamics of the interactions. Both aspects must be addressed in developing agent-based models. By that, it is important to mention that agent-based systems are decentralized systems and only *local information* is available to an agent [11]. Agents typically interact with a subset of other agents, termed the agent's *neighbors*. How agents are connected to each other is generally termed an agent-based models *topology* or connectedness. A topology describes who transfers information to whom [12]. Typical topologies include a graph-connected network, or a spatial rectangular or hexagonal grid, like it has been used in terms of this work and it is described in the next chapter.

2.3 Cellular Automata

Originally, spatial agent-based models were implemented in the form of cellular automata (CA). Around 1950 CA were introduced by Stanislas Ulam, John von Neumann and Konrad Zuse. Ulam simulated the growth of pattern in two and three dimensions. John von Neumann proposed a self-reproducing cellular automation which at the time realized a universal *Turing machine*. A Turing machine is an automation with only three operations (read, write, move on), that is capable of manipulating information stored on a tape [9]. All basic arithmetic operations can be implemented on these machines and with them other, more complex operations and programs can be produced. Later three dimensional CAs proved to be interesting for simulation of gas- or liquid-distributions in space. Conways Game of Life (Gardner, 1970) is also a good example of CA. It uses two-dimensional cellular automata with two possible states (living and dead) combined with a set of rules to demonstrate the self-organization of living cells. It is interesting to observe, that a lot of two-dimensional patterns can emerge from the implementation of a small set of simple rules. However, it has to be mentioned that cellular automata can hold more than two states and the rules of movement and neighborhood can be adapted in different ways.

According to Gladrow [10], following points have to be considered when defining a CA:

- *Geometry of cells* - the CA consists of equal (geometrical) cells that are arranged in a regular lattice
- *Neighborhood definition* - determines which cells are influencing each other
- *Cell states* - A finite number of discrete states is defined. Every cell can assume one of those states. The transition between states is determined by the transition rules.
- *Transition rules* - describe how the cells change their states. These rules depend on the state of the neighboring cells and on the state of the cell itself. One set of transition rules valid for the whole CA over the run time.

Two-dimensional cellular automata

Two neighborhood rules used for 2D CA are so-called *Von Neumann neighborhoods* and *Moore neighborhoods* [10].

Von Neumann neighborhoods of range *r* are defined by

$$N_{i,j}^{vN} := (k,l) \in L || k-i| + |l-j|| \leq r \tag{2.5}$$

and *Moore neighborhoods* of range *r* by

$$N_{i,j}^{M} := (k,l) \in L | k-i| \leq r \wedge |l-j| \leq r \tag{2.6}$$

In following, we will assume a CA with Moore neighborhood (Figure 2.6). And let *n* be the number of possible states that a cell can take on in accordance to the transition rule Φ. Φ is updating the state of every cell at every time step. The new value of c_{ij} with location *i*, *j* only

depends on the state of the cell itself and the states of its neighbors. Thus the change of $c_{i,j}$ can be expressed after one timestep t $\rightarrow$ t+1 through

$$c_{i,j}^{t+1} = \Phi(c_{i,j}^t, c_{i-1,j-1}^t, c_{i-1,j}^t, c_{i,j+1}^t, c_{i,j-1}^t, c_{i,j+1}^t, c_{i+1,j-1}^t, c_{i+1,j}^t, c_{i+1,j+1}^t) \tag{2.7}$$

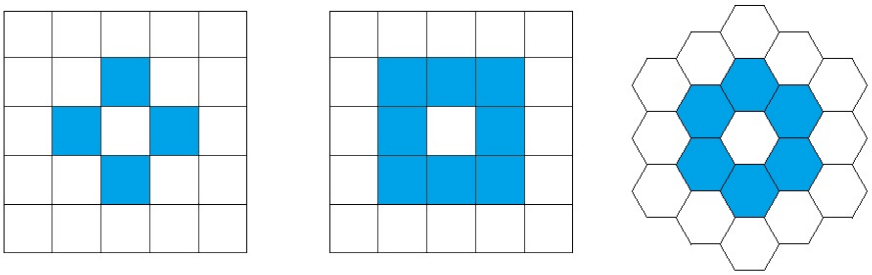

Figure 2.6: Von Neumann (left), Moore (center) and hexagonal (right) neighborhoods of range 1

The FHP Lattice-Gas Cellular Automata

As described in detail in chapter 3.2, two dimensional cellular automata are used for simulation of spatial contacts in the presented agent based model. Due to the advanced characteristics in the simulation natural movement (gases, fluids, as well as pedestrian dynamics [**?**]), the FHP Lattice-Gas CA was chosen for concrete implementation as an extended form of the 2D CA.

According to Gladrow [10], the main features of FHP lattice-gas cellular automata can be summed up by following points

- The underlying regular lattice shows *hexagonal symmetry*.
- *Nodes* (sites) are linked to six nearest neighbors located at the same distance with respect to the central node.
- The sectors linking nearest neighbour nodes are called *lattice vectors* or *lattice velocities* if i=1,...,6

  $$c_i = (cos\frac{\pi}{3}i, sin\frac{\pi}{3}i) \tag{2.8}$$

  with $|c_i| = 1$ for all i.
- A cell is associated with each link at all nodes.
- A cell consists of 6 lattice vectors

- Each lattice vector can be occupied by at most one particle *(exclusion principle)*.
- All particles have the same mass and are indistinguishable.
- The evolution in time proceeds by an alternation of *collision* C and *streaming* S (also called propagation), where ϵ is called *evolution operator*

$$\epsilon = S \circ C \tag{2.9}$$

- The collisions are strictly *local*, i.e. only particles of a single node are involved.

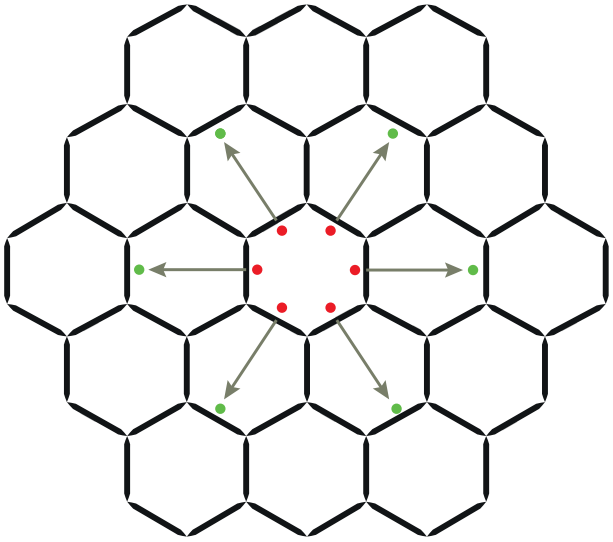

Figure 2.7: The hexagonal lattice of FHP LGCA model is decomposed of symmetrically ordered triangles. Particle movement from lattice vectors $c_{1..6}$ is visualized by gray arrows, indicating particles at timesteps t (red dots) and $t + 1$ (green dots).

If no collisions occur, every particle inside the FHP model has exactly one lettice vector in one specific cell to occupy in a future time step, as it is shown with the help of gray arrows at Figure 2.7. They indicate the transition from particles at red positions at time step t to their propagation destination at green particle position at a future time step $t+1$ [10]; this rule is used at all cells and particle vectors of the cellular grid and iterated together with the collision rules for every time step of the simulation.

If particle reach the boundary of the grid they will be re-spawned on the opposite side in either horizontal or vertical direction as if they would be connected to each other forming the grid to a cylinder at each direction. As this way to implement boundary contacts was proposed by Gladrow [10], other possibilities to implement boundary conditions of particles movement will be discussed in Chapter 3.3.

An important point for preserve the symmetry is the following of the collision rules (figure 2.8) which have to be non-deterministic in order to prevent spatial reflections [10]. Because generating random numbers is a computational time consuming process, a pseudo-random choice of the rotational sense is allowed by taking only one random number per time step for all collided nodes. As shown in figure 2.8 there are 5 collision rules in total, dividing into two-, three- and four-head collisions. One can distinguish between different FHP variants, that use either all of the given rules, or just a necessary subset of two- and three-head collisions. To be exact, the use of rules d) and e) is not necessary when the model is not aiming to reproduce physical dynamics of gases or fluids (the automaton using this subset of rules is called FHP-I). This is the reason why we do not implement it in the subsequently presented epidemic model, in respect to the reduction of computing time.

Frisch, Hasslacher and Pomeau showed that this lattice-gas cellular automata model over a lattice with a larger symmetry group than for the square lattice yields in the incompressible *Navier-Stokes equation*, which can described as:

$$\frac{\partial u}{\partial t} + (u\nabla) = -\nabla P + \nu\nabla^2 u \tag{2.10}$$

with the continuity equation

$$\nabla u = 0 \tag{2.11}$$

and ∇ being the nabla operator, **u** the flow velocity, P the kinematic pressure, ν the kinematic shear viscosity and ∇^2 the laplacian operator. Navier-equation is nonlinear in the velocity **u** and thus in general analytically not solvable-save for a few exceptions [17] This question is connected to the question of boundary condition. Because of this complexity, numerical methods need to be applied in order to achieve computation of the equation. The discovery of the symmetry constraint of lattice-gas cellular automata was the start for a rapid development of lattice-gas methods. Within the following years many extensions, generalization and a wide range of applications were proposed.

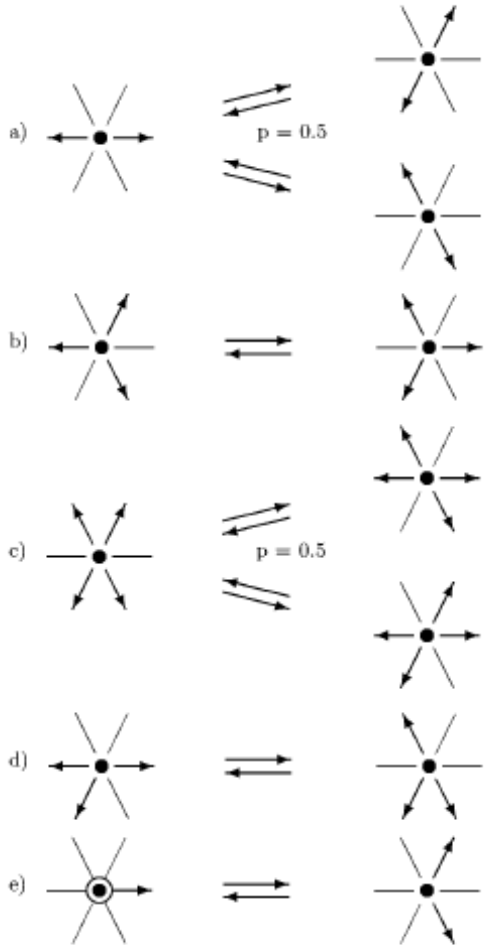

Figure 2.8: Collision rules according to Gladrow [10]

CHAPTER 3

Agent Based Epidemic Simulation on Spatial Grids

The topic of the following chapters is the combined use of agent based modeling, SIR epidemics and cellular automaton dynamics in the purpose to create a new type of epidemic model. After a short overview of previous approaches in this area I will explain our variant of the epidemic model in detail and introduce a parameter study observing the core parameters of the model. Afterwards, a visualization is presented to show the spatial characteristics of the model. In the later chapters, the random contact model will be introduced and the epidemic spread of both models will be compared to discuss the convertibility of the spatial grid model.

3.1 Previous approaches

One of the first manuscripts discussing the need of a spatial component in epidemic simulation was published by Yakowitz et al [15] in the year 1990. The presented work extended classical epidemiological calculations with a probabilistic model called “random cellular automaton", defining the movement rules of a 2D-cellular automaton and infection dynamics of a SIR model as a stationary Markov process. Yakowitz et al motivate their afford with following citation: “Most of the past research on the spread of disease assumes that the people amongst whom an infection is spreading mix homogeneously. Whilst this is appropriate for certain situations it is inappropriate for a disease spreading over a large area."
To underline the importance of the spatial component, they created a simulation study to emphasize differences between two different spatial configurations of grid particles at the start of the simulation: in the first one, the particles are distributed homogeneously over both axes of the grid, while on the second setup, they reside agglomerated in the middle, forming a “cluster". After a simulation has been computed at each of the grid configurations with exactly the same parameter settings and computing duration, the simulated results of both grid setups were compared to each other. It is shown that the agglomerated distribution of particles leads to a higher

contact rate at total amount of particles than the homogeneous distribution, because contacts between particles happened considerably more often during the given simulation time.

Lattice gas cellular automata in combination with SIR type models were presented by Fukś et al [16] in 2001 with the aim to observe vaccination strategies at SIR type epidemics. In this computational study, the authors presented the effect of vaccination shortness (if the amount of vaccinated individuals is less then the total amount of susceptible population) when comparing two different vaccination strategies. In the first one, so called “uniform strategy", a determined number of individuals were selected randomly from the whole population. In the “barrier strategy" on the other hand, a ring was formed around the susceptible group and only the most outer individuals were infected. With the help of propagating LGCA particles, it has been shown that the barrier strategy is more effective when a limited amount of vaccination is available. Due to this observations, Fuks et al discussed the importance of the usage of a spatial epidemic simulations, stating classical SIR models “abandon the assumption of homogeneous mixing and allow to study the geographical spread of epidemics" .

Štefan Emrich [17] presented a similar approach of an epidemic simulation, where agents contacts were calculated at discrete time and space on a hexagonal CA grid, aiming to provide a model of Influenza infection upon agents from a homogeneous population. In this model, agents with different properties were put together in households, workplaces or leisure time places and organized randomly on the CA grid for a defined simulation time. The obtained contacts were used to calculate the epidemic spread in the population. Because this model was a motivation for the actual approach, the detailed connection between LGCA, SIR and Agents will be explained in the next chapter.

3.2 Cellular automaton based model

Our model is built from interaction of three different parts or modules. The first module consists of the agent based paradigm described in chapter 2.2. It supplies the model with a bottom up method to view and simulate the population in our epidemic simulation. In detail, every person or “agent" used in the simulation is defined by a set of variables and parameters that are important to describe his state, behavior and influence on other agents. For example, a necessary detail for an agent in an epidemic simulation could be the age of a person, as it represents a dependent variable to the infection probability, when the likelihood of an infection is different at a certain age.

The second module used in our model is the two dimensional FHP lattice gas cellular automaton, described in chapter 2.3. This algorithm is used to provide a spatial component to the agents in the simulation. By definition, the FHP LGCA provides a set of propagation rules to equal particles, so this paradigm gets extended by connecting each of the particles to a specific agent ID. This results in a simulation of discrete spatial movement of the artificial agents, where the collision rules of the cellular automaton are very helpful at the determination of epidemic contacts between the agents. Whenever two particles collide on the lattice grid, in terms of having one node occupied with $i > 2$ velocities, the agent module of the model gets informed about

a potential contact. In this way, further calculation of the infection probability get triggered, and in the next discrete time step of the simulation, the agents continue their movement as defined by the particle rules of the FHP LGCA with a potentially changed health status.

The third module used in our model is the SIR-model, sitting on “top" of the two previously described modules. The function of the SIR model is to supply the simulated agents with dynamics of infection and health recovery, which should lead to a global development of an epidemic, as explained in chapter 2.1. The first step in connecting the SIR-model to the global epidemic model is the extension of personal parameters of every agent with a health status - ether susceptible, infected or recovered. Now, when agents move discretely in the space with the initial health state “susceptible" and eventually collide with other agents, they can get infected, by changing their health state due to a computed infection probability. In the other case they can re-convalesce from the infection and change their health status to “recovered". In this way, the connection of the modules leads to the desired simulation of epidemic spread and provides the possibility for extension at every level.

However, in the context of this specific work, we want to focus on the second module, the cellular automaton which represents the connection between agents and infection dynamics. For sake of simplicity, let us assume the agents of following simulations do not have specific properties, so they are totally equal to each other beside of their unique identification number and individual health state. In order to describe a set of default simulation parameters the parameters of the SIR model, as explained in chapter 2.1, were set to following values:

- *Infection Probability* - 2%
- *Duration of Infection* - 10 timesteps, starting from the timestep of infection

When defining the parameters of the LGCA module, the most important parameter for setting up a lattice automata grid is the *density* or ρ. This is defined by:

$$\rho = \frac{N_{particles}}{6 * dim_x * dim_y} \tag{3.1}$$

Where dim_x and dim_y are the width and height of the grid and $N_{particles}$ is the number of particles (corresponding to agents) on the grid. If the density is known, it can be used to calculate the (quadratic) size dim of the cellular automaton.

$$dim = \sqrt{\frac{N_{particles}}{6 * \rho}} \tag{3.2}$$

In the future parameter study, the default amount of agents $N_{particles} = 8000$, and the default density $\rho = 0.533$. Thus, the quadratic size of the cellular grid is given by $dim = 50$. The resulting SIR development of the default parameter set can be seen in figure 3.1,

3.3 Parameter study

We call the basic simulation run of the epidemic model with SIR parameters as defined in Chapter 3.2

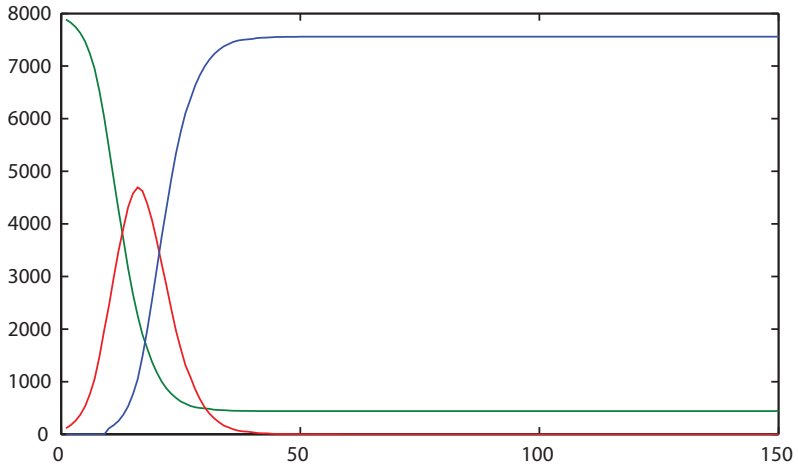

Figure 3.1: SIR development of the basis parameter set, green=susceptible, red=infected, blue=recovered

- *Density* - 0.533 (Resulting from 8000 Agents on an 50x50 lattice grid)
- *Starting position of infected agents* - Infected agents are randomly distributed among the whole grid
- *Starting position of susceptible agents* - Susceptible agents are randomly distributed among the whole grid
- *Total simulation time* - 150 time steps

Regarding the discrete space of the lattice grid of height *dimx* and width *dimy*, we have to distinguish between two modifications of the *particle rebound* of the cellular automaton model:

- A model implementing interconnected space in both directions, realized by an overflow at a given side by creating a connected boundary (imagined by transformation of a plane into a cylinder). So, an particle or agent with actual coordinates *x,y* moving forward in positive horizontal direction will contact the lattice boundary *(xdim,y)* and will be re-spawned at the cell *(0,y)* in the next timestep, as well as an agent moving in the positive vertical direction at position *(x,ydim)* will re-spawn at cell *(x,0)*.
- A model implementing a particle or agent “rebound" at the boundary of the cellular grid. Although there are various implementation of the rebound mechanism, the model used in this simulation rebounds particles in the node coordinate *x*=*xdim* from position c_4 to

c_1 and from c_1 to c_4 at node coordinate *x=0*. The rebound at the node *y=ydim* sets the particles at position c_5 to c_2 and vice versa, as well as the diagonal rebound at position c_3 towards the contrary position c_6.

All of the presented simulations were implemented including particle rebound on edges of the cellular grid to allow the simulation of complex starting position of the infected population, as for example an isolated start morphology at one corner of the grid.

In the first part of the parameter study, we focus on the variation of *cellular grid density* and its effect on the development of the SIR infection. As it can be seen in Figure 3.2, a lower density causes a slower spread of infection and a lower total percentage of infected population. In comparison to the basic simulation run, higher grid densities (0.66 and 0.8) lead to a slight increase of total percentage of infection, although the shape of the development does not change significantly.

In the second part of the study, the parameter of *infection probability* has been varied to values below and above the basic simulation setting of 2%. In Figure 3.3 it can be seen that an infection probability of 0.5% is not enough to cause an epidemic spread with a major amount of recovered agents at the end of the simulation.

The variation of the *infection duration* at Figure 3.4 shows similar results as the variation of the *infection probability* parameter, where the settings below the basic simulation induce a lower amount of infected individuals and a value above the basic simulation generates a greater and faster epidemic spread.

Interestingly, the change of the *starting position* of the infected population in simulations as described in Chapter 3.2 causes either a delay of infection when placed in the corner or a shallower development of the infected population, when placed clustered in the middle of the cellular grid, as shown in Figure 3.5. In both cases, the total amount of infected agents is lower than in the basic simulation run (the susceptible population is randomly distributed at every case, the particle position in the node is determined randomly).

3.4 Visualization of an epidemic spread

For a better understanding of the presented concept we created a visualization of an epidemic spread simulated by the *cellular automaton based model*. The plots presented below show a picture of the hexagonal grid used in the CA based model (chapter 3.2) at 9 different discrete time steps. The cells may be occupied by agents with a variable health status which changes due to definitions of the SIR model (chapter 2.1). The position and health status of an agent at the given time step is visualized by a colored pixel in the plot, which can be either green (*susceptible state*), red (*infected state*) or blue (*recovered state*). All presented visualization in Figures 3.7 to 3.14 are representations of the CA-based epidemic model executed with *basic simulation parameters* as defined in 3.2 but with variations of grid density and starting positions of the infected population.

The first three plot groups of this chapter show the epidemic spread starting with a randomly distributed infected agent population at the beginning of the simulation (as it is the case at the basic simulation parameters) and a cellular density set to 0.26 (Figure 3.6), 0.53 (Figure 3.7) and

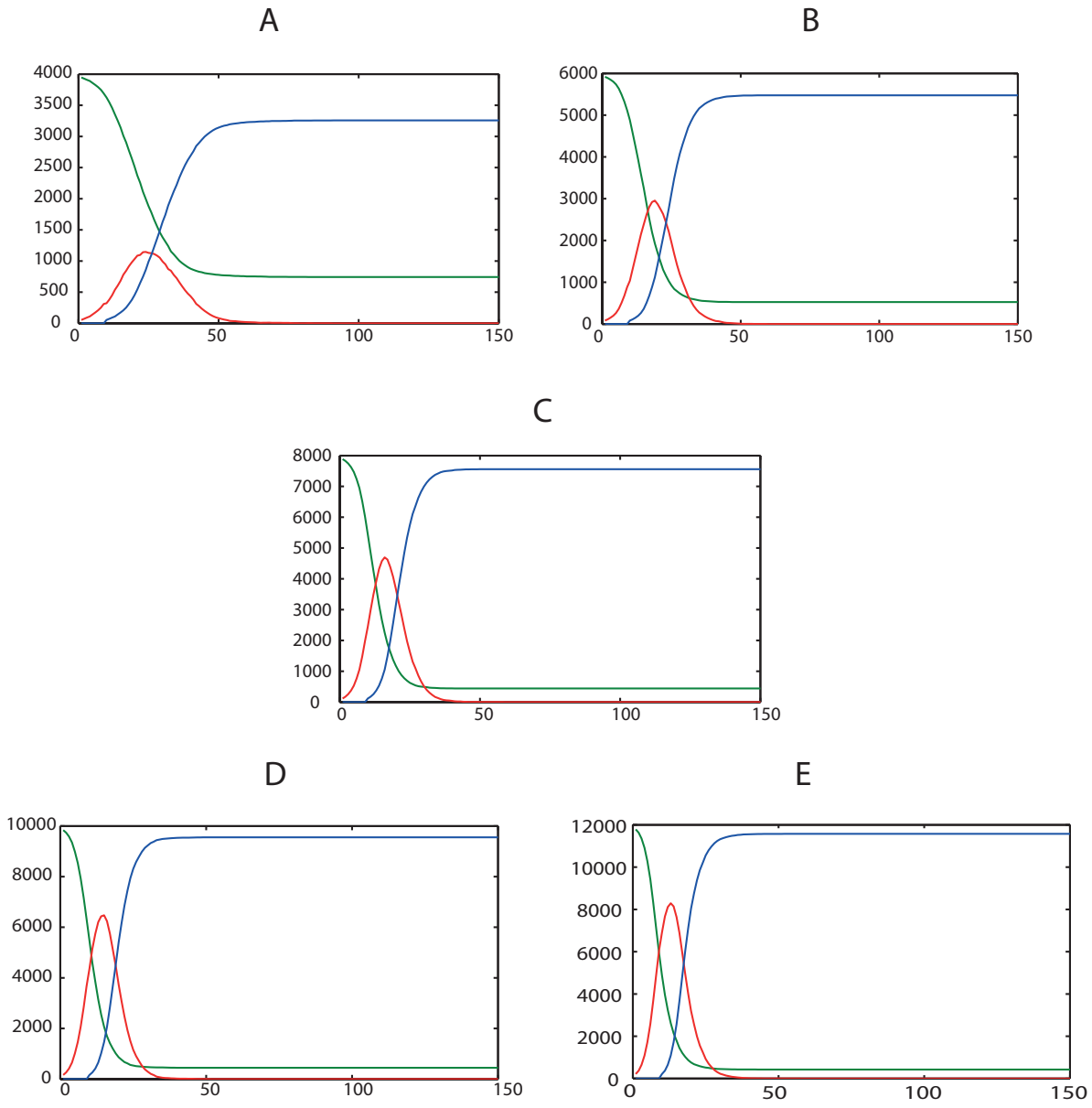

Figure 3.2: Variation of the cell density, resulting from a different amount of agents at an equally dimensioned grid: A: 0.26 (4000 Agents), B: 0.4 (6000 Agents), C: Basic simulation run, D: 0.66 (10000 Agents), E: 0.8 (12000 Agents)

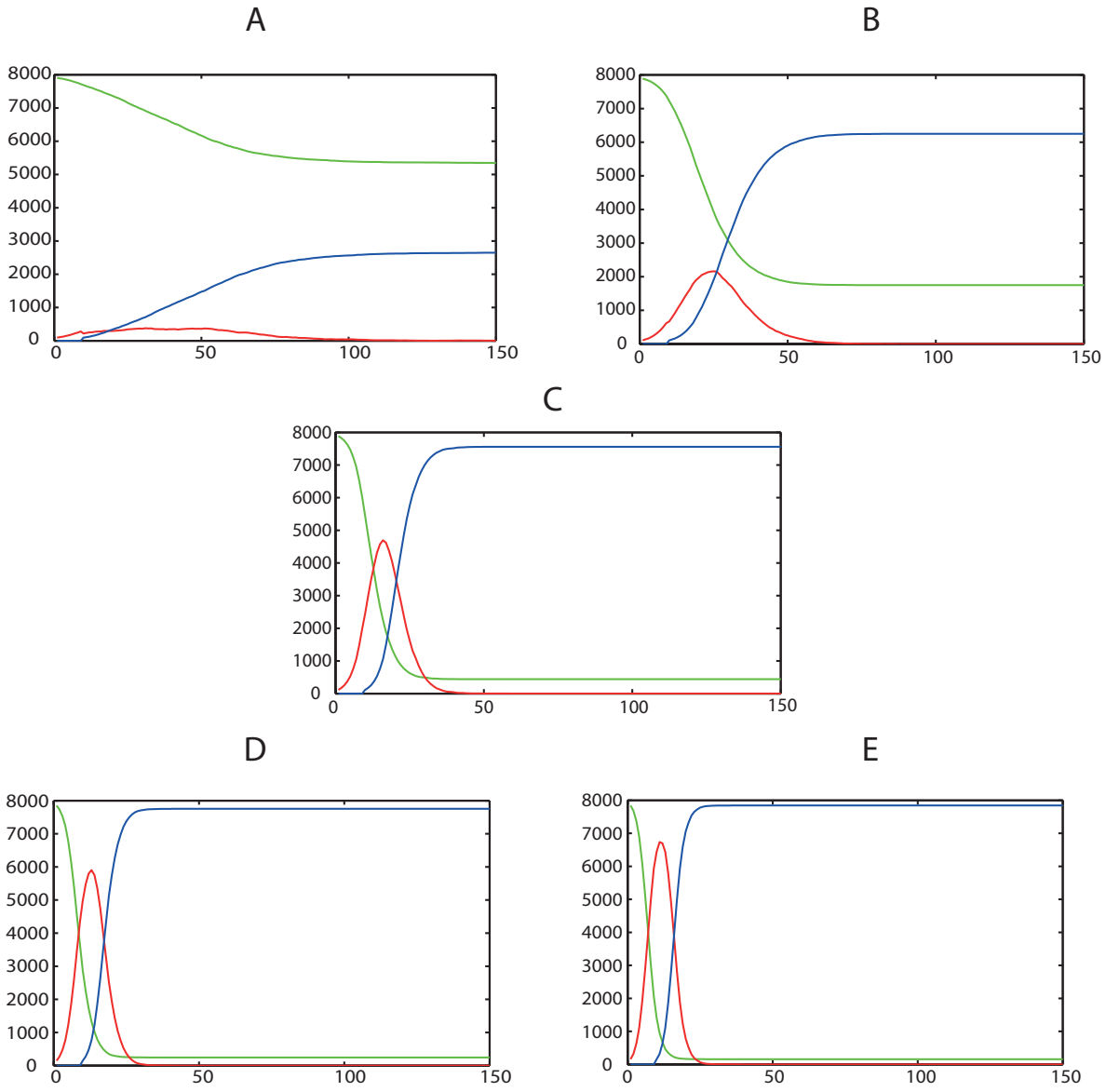

Figure 3.3: Variation of infection probability of a susceptible agent: A: 0.5%, B: 1%, C: Basic simulation run (2%), C: 3%, D: 4%

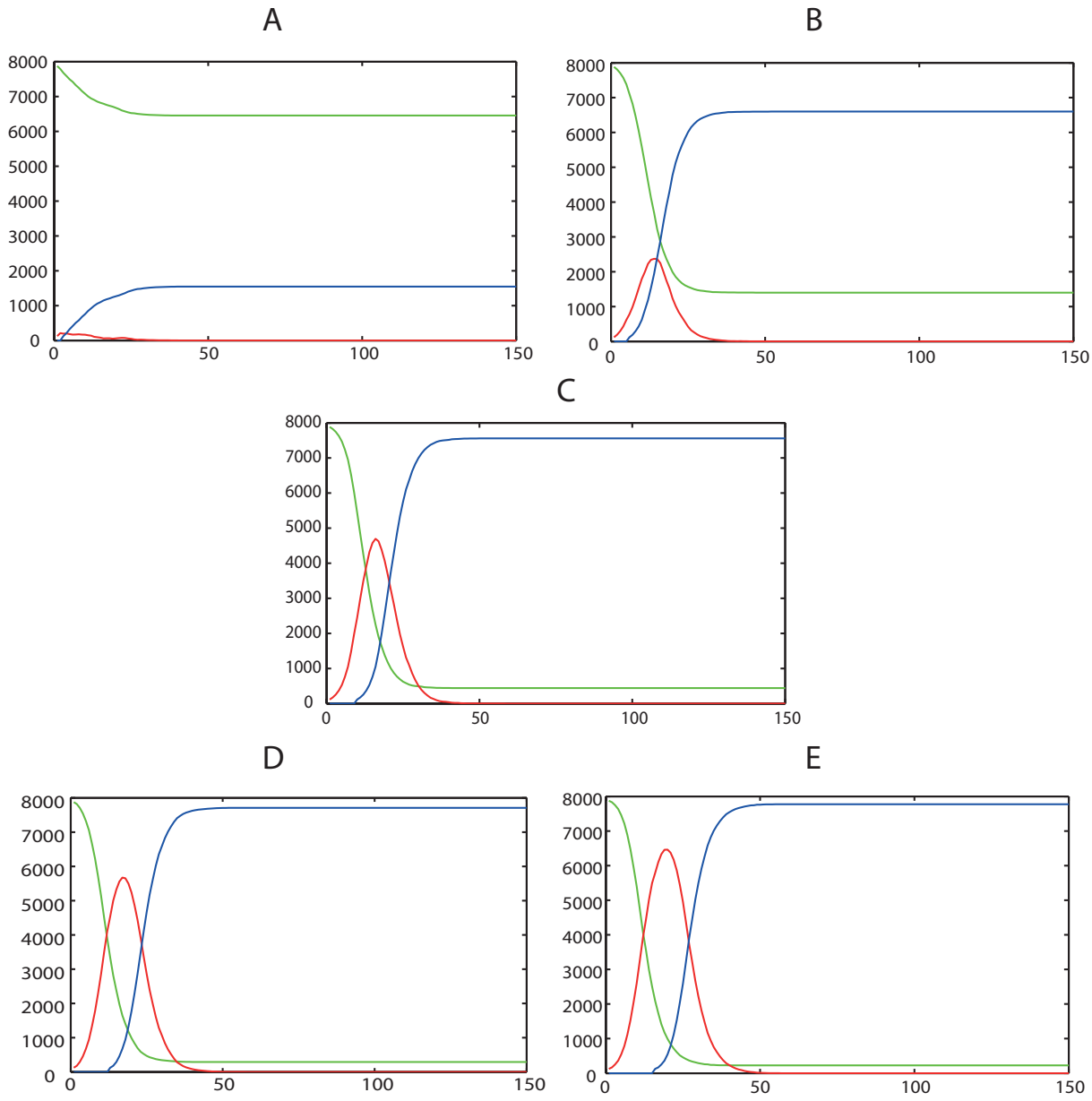

Figure 3.4: Variation of infected duration of an infected agent: A: 3 timesteps, B: 6 timesteps, C: Basic simulation run (10 timesteps), D: 13 timesteps, E: 16 timesteps

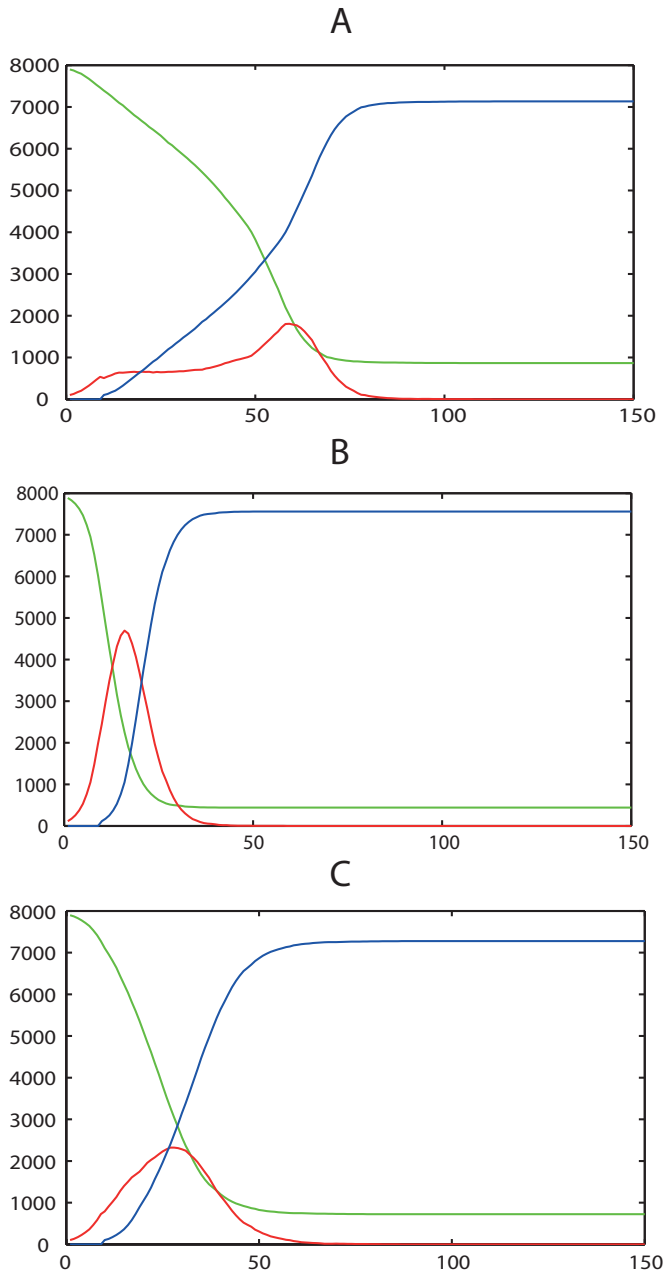

Figure 3.5: Different starting position of the infected population at $t0$: A: Infected population agglomerated in the lower left corner, B: Basic simulation run (Random distribution of infected agents on the whole grid), C: Agglomeration in the center of the grid

0.8 (Figure 3.8). It should be remarked that the cellular density is changing due to an increasing amount of agents on an equally dimensioned grid in all simulation cases. As we compare Figure 3.6 with Figure 3.8 we can observe a slower and more shallow spread of the infection at the cellular grid with low density. This effect may happen because the initially infected agents encounter susceptible agents more unlikely at this density level (only every fourth lattice vector is occupied at $\rho = 0.26$). This distinction can be observed more precisely at the identical time step of both simulations (subplots 3.6C and 3.8C). The starting configuration represents an infection outbreak at undefined regions of a homogeneous population which could be caused by a mutation of a virus that infects random receptive individuals.

Figures 3.9 to 3.11 show the same variation of grid densities (0.26, 0.53, 0.8) but with a starting configuration of infected agents at the right upper corner of the cellular grid. This starting configuration could represent an infection hazard from a certain direction of an isolated area (or country, when interpreted at a high scale), similar scenarios can be used for tests and validations of vaccination barriers. Again, infection on lower grid density propagate much slower than on higher densities as can be seen in the SIR development graph in subplots 3.9J and 3.11J. It should be emphasized, that these SIR graphs present a shallow development of the infected population, because these agents were placed on the grid *non-homogeneously* at the start of the simulation.

The density is diversified in the same manner at simulation setups of Figures 3.12 to 3.14 while the group of initially infected agents (1 % of the total population) starts in agglomerated state at the center of the cellular grid. One can recognize the difference of these infection spreads; The spread at Figure 3.12 propagates slowly, so infected individuals can overcome a region without interfering with all susceptible agents in their surrounding. On the contrary, the infection spread at high density (Figure 3.14) represents a mass infection, spreading in circular form with a constant speed and infecting all susceptible individuals at the outer barrier. Beyond that, the percentage of recovered agents is much higher in the center on the barrier, when comparing time step 17 at subplots 3.13F and 3.14F. The ellipsoid form of the epidemic population spread at subplots B-F at Figures 3.12 to 3.14 is caused by artifacts of the hexagonal plot producing the cellular lattice as shown in Figure 2.7. Because the distance to the future propagation step of particles at particle velocities c_2 and c_5 is projected to a more distant x/y point then at velocities $c_{1,3,4,6}$, the plot is optically distorted in diagonal directions. When we analyze coordinates of propagating cells, the spread of epidemic agents from a central starting configuration is expanding homogeneously at equal speed in all directions and is *rotational invariant* if the corresponding cellular grid is turned by $90°$, $180°$ or $270°$.

3.5 Random Contact Model

In this chapter we present an alternative to the cellular automaton based model known from Chapter 3.2 which uses the same agent based methodology and SIR infection dynamics as the previous model, but differs in the mode of agent contact finding. More precisely, the presented cellular automaton concept which has been used conveniently for simulating moving and contacting agents is being exchanged with a random generator assigning agents to each other with given probabilities and providing the SIR module with these randomly generated epidemic con-

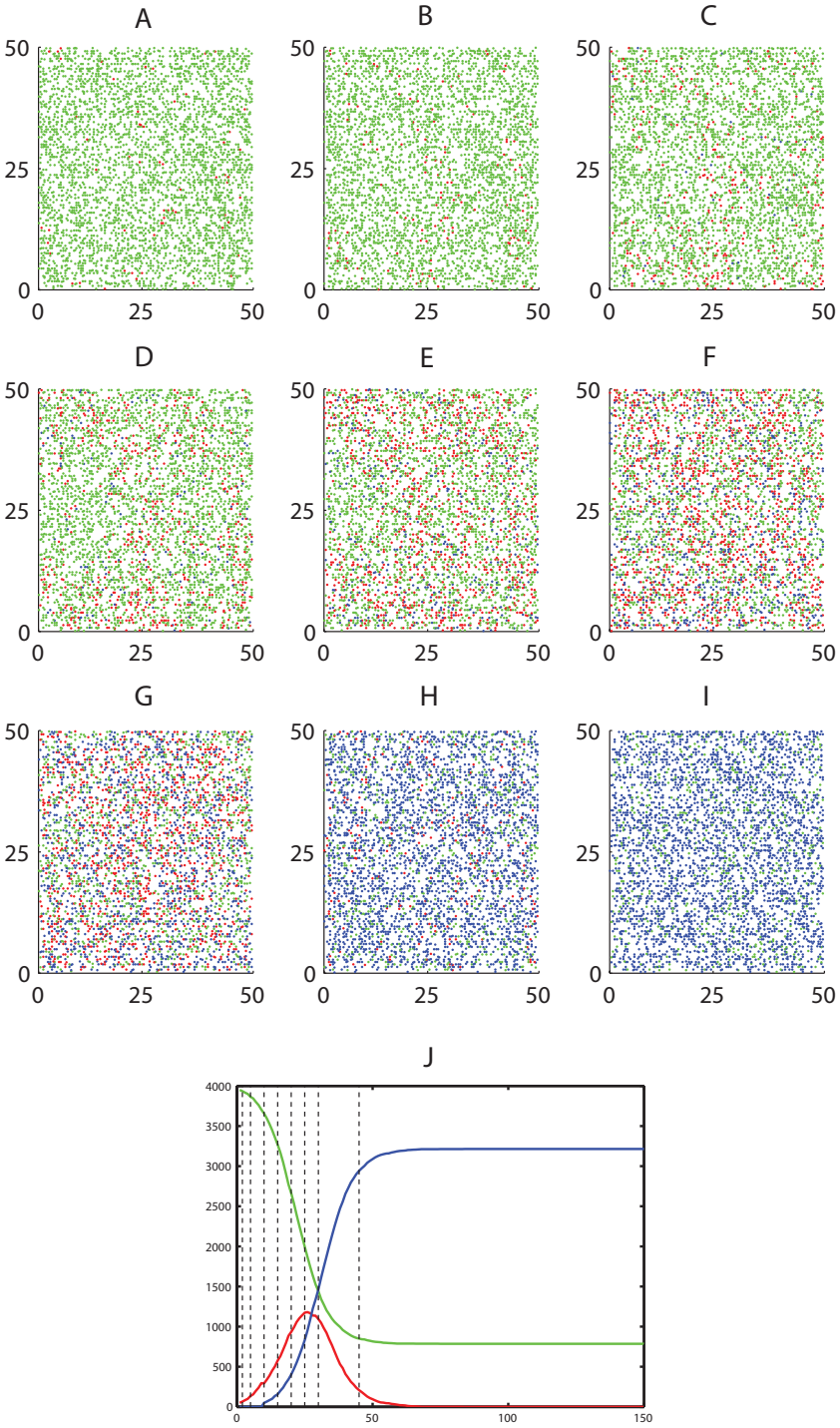

Figure 3.6: Visualization of epidemic spread of randomly distributed infected agents at $t0$ and cellular density of 0.26. Time steps: 1(A), 5(B), 10(C), 15(D), 20(E), 25(F), 30(G), 40(H), 150(I)

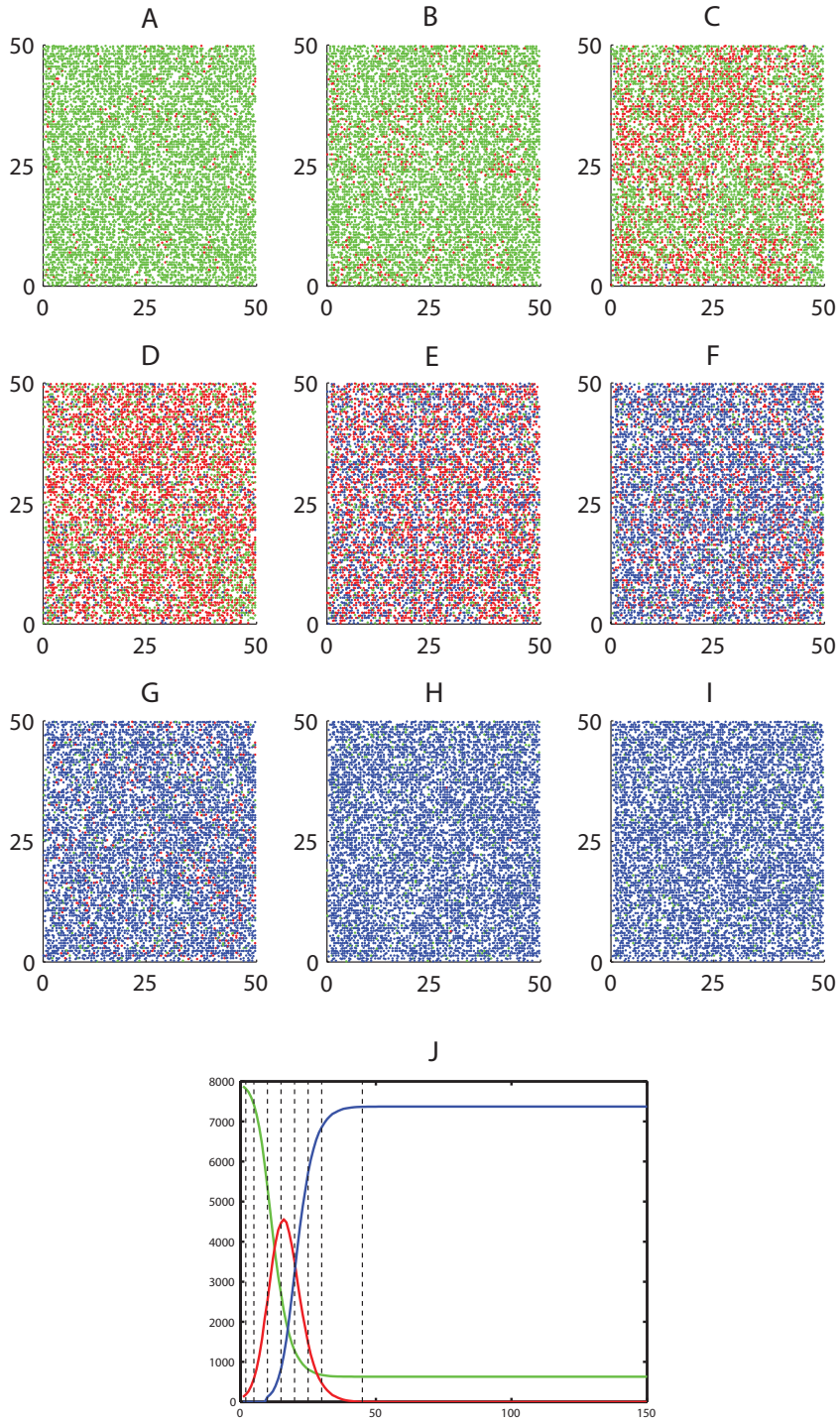

Figure 3.7: Visualization of the epidemic spread with basic simulation parameters and random distribution of infected agents at $t0$ at simulation time steps 1(A), 5(B), 10(C), 15(D), 20(E), 25(F), 30(G), 40(H), 150(I) and the corresponding plot of agents at every health state over time

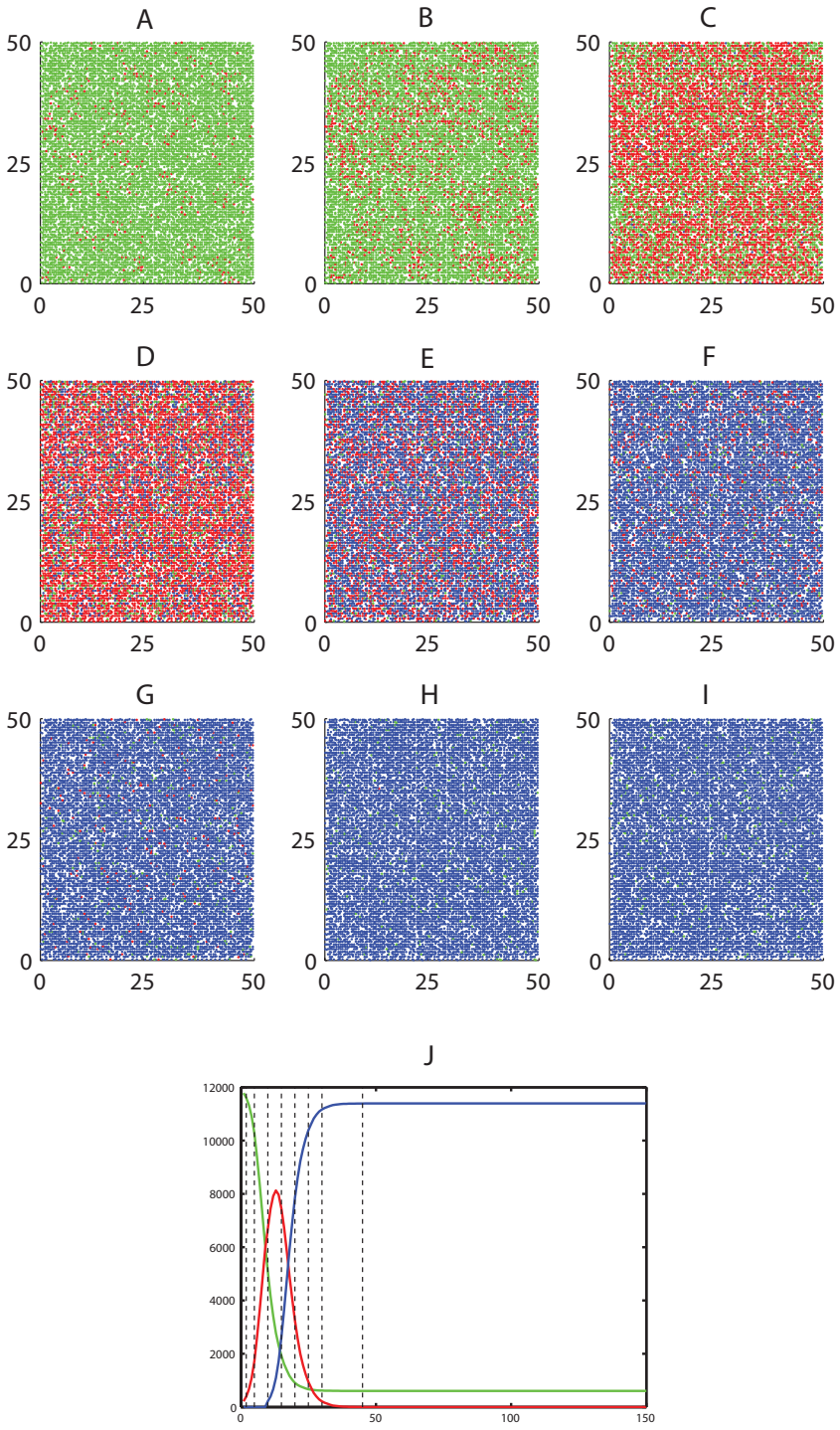

Figure 3.8: Visualization of epidemic spread of randomly distributed infected agents at $t0$ and cellular density of 0.8. Time steps: 1(A), 5(B), 10(C), 15(D), 20(E), 25(F), 30(G), 40(H), 150(I)

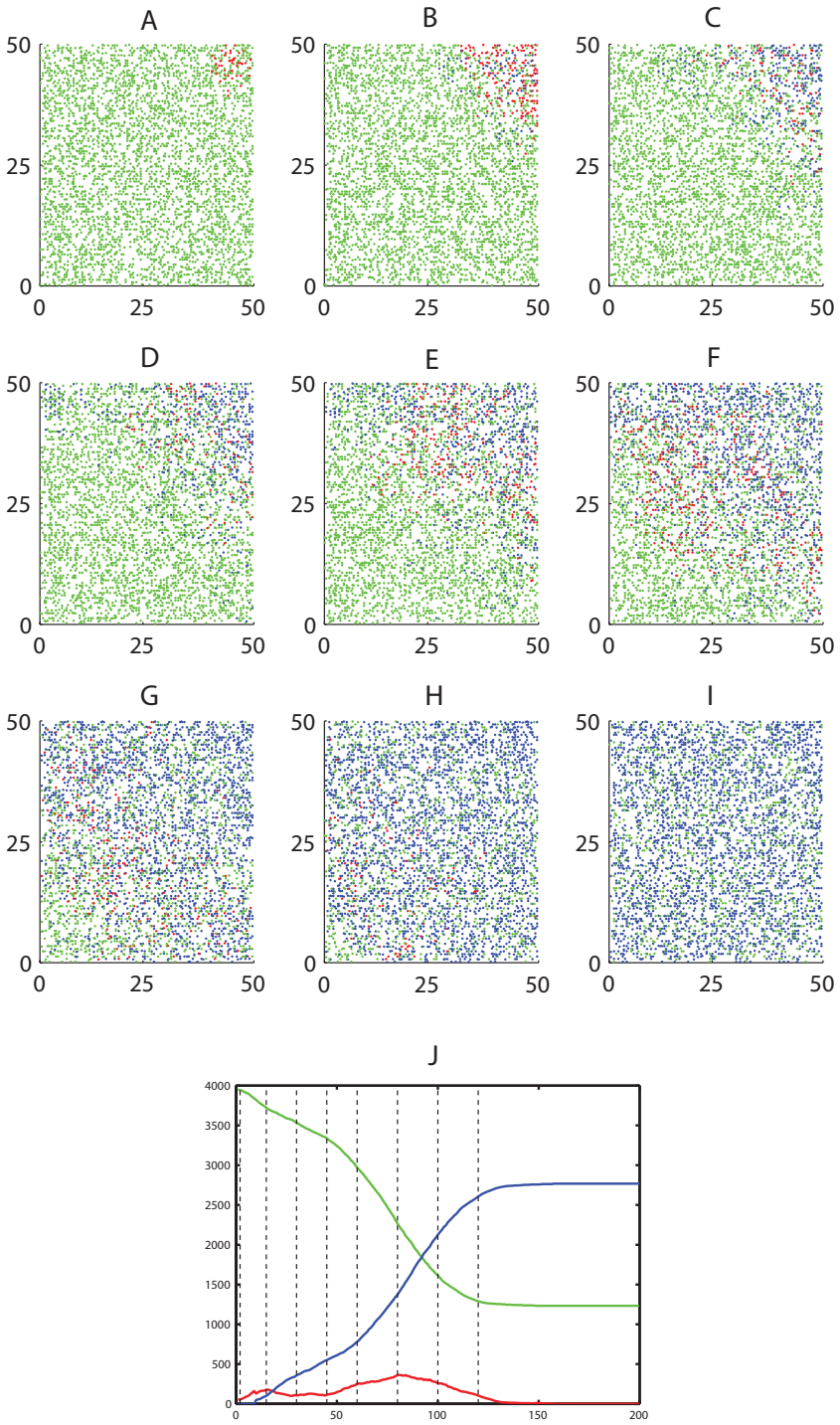

Figure 3.9: Visualization of epidemic spread of infected agents starting in one corner of the grid at begin of the simulation and cellular density of 0.26. Because of the slow development of epidemic spread at this configuration, the simulation time was increased to 200. Time steps: 1(A), 15(B), 30(C), 45(D), 60(E), 80(F), 100(G), 120(H), 200(I)

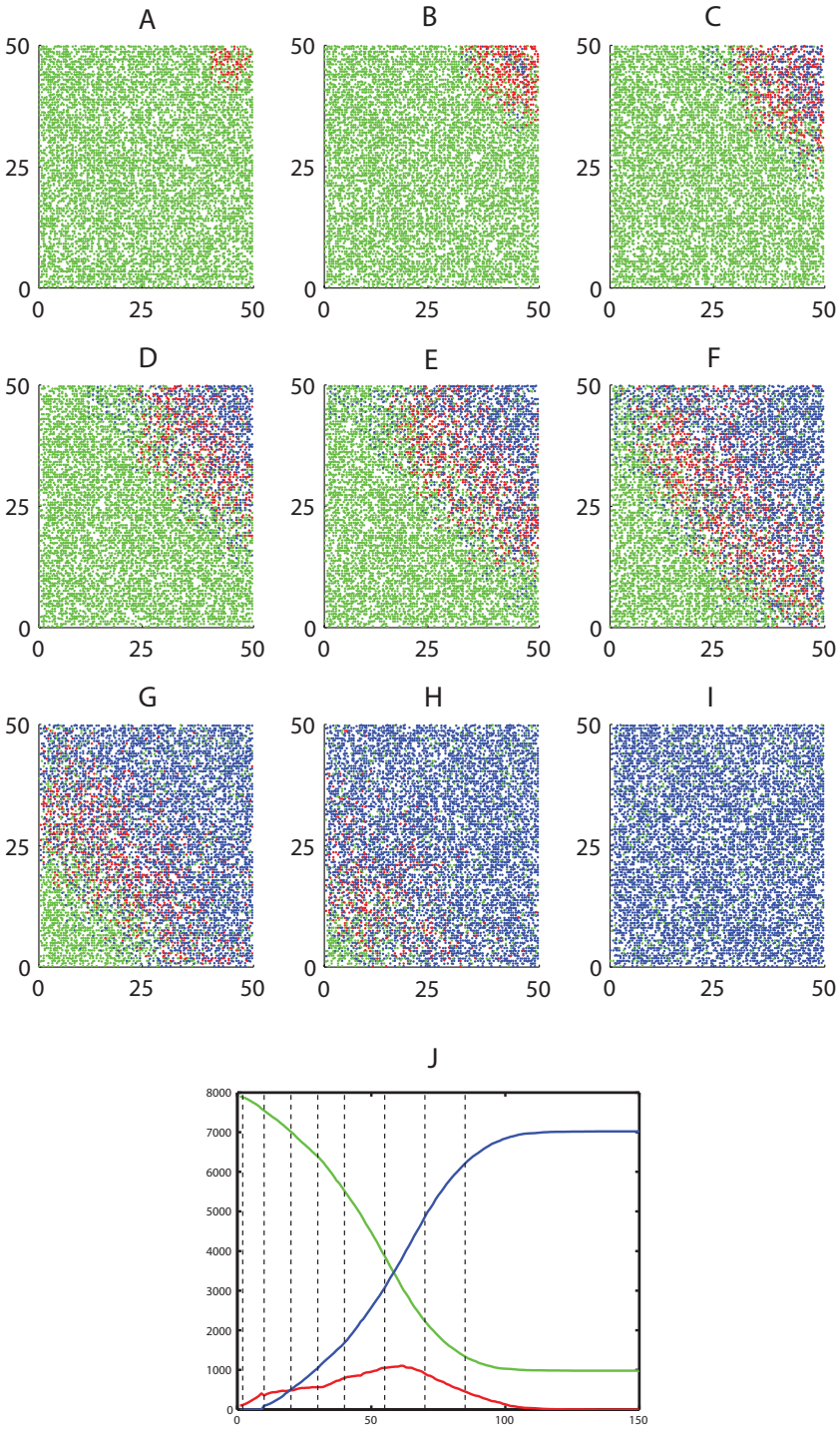

Figure 3.10: Visualization of epidemic spread of infected agents starting in one corner of the grid at begin of the simulation. Density is corresponding to the basic simulation parameters (0.53). Time steps: 1(A), 10(B), 20(C), 30(D), 40(E), 55(F), 70(G), 85(H), 150(I)

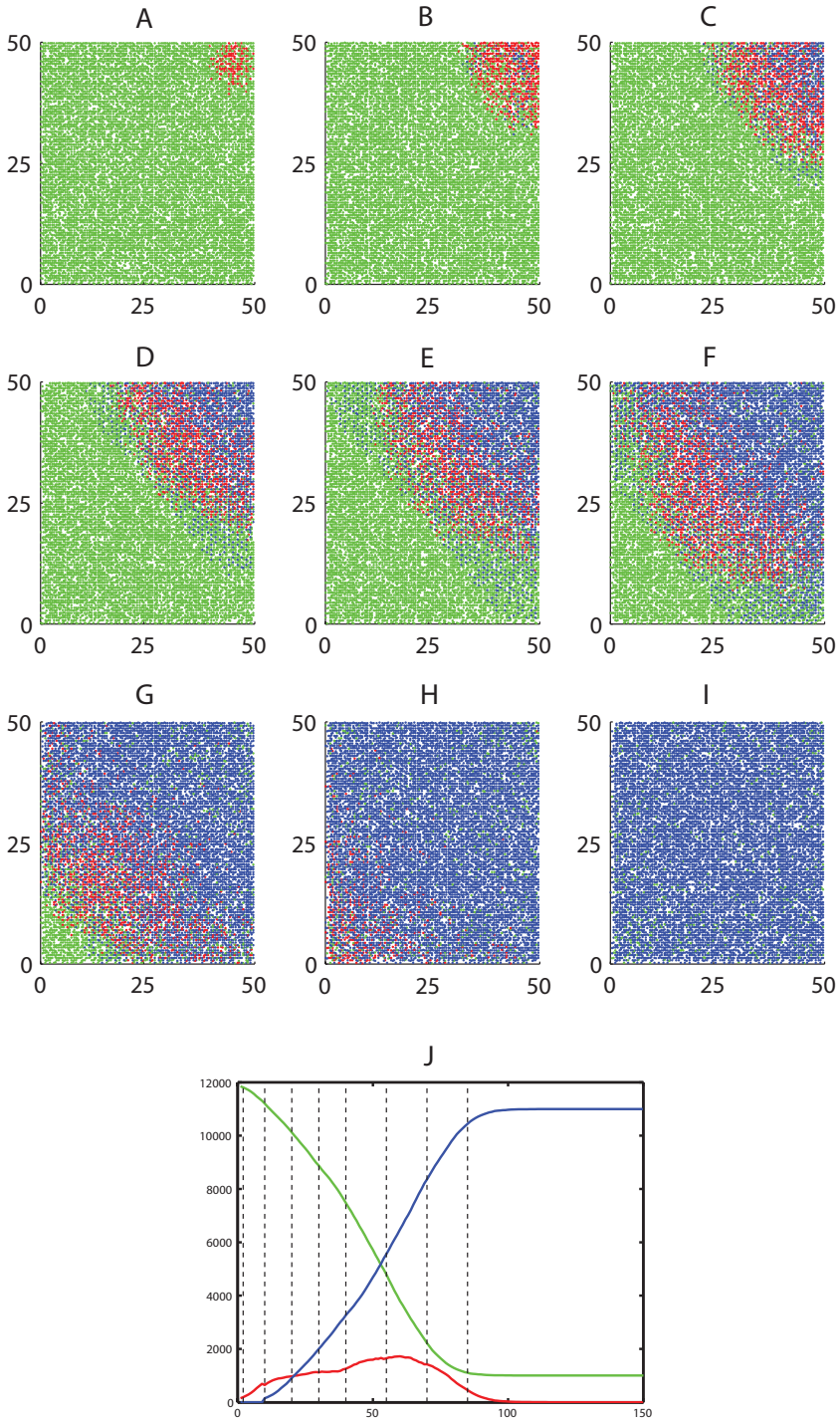

Figure 3.11: Visualization of epidemic spread of infected agents starting in one corner of the grid at begin of the simulation and cellular density of 0.8. Time steps: 1(A), 10(B), 20(C), 30(D), 40(E), 55(F), 70(G), 85(H), 150(I)

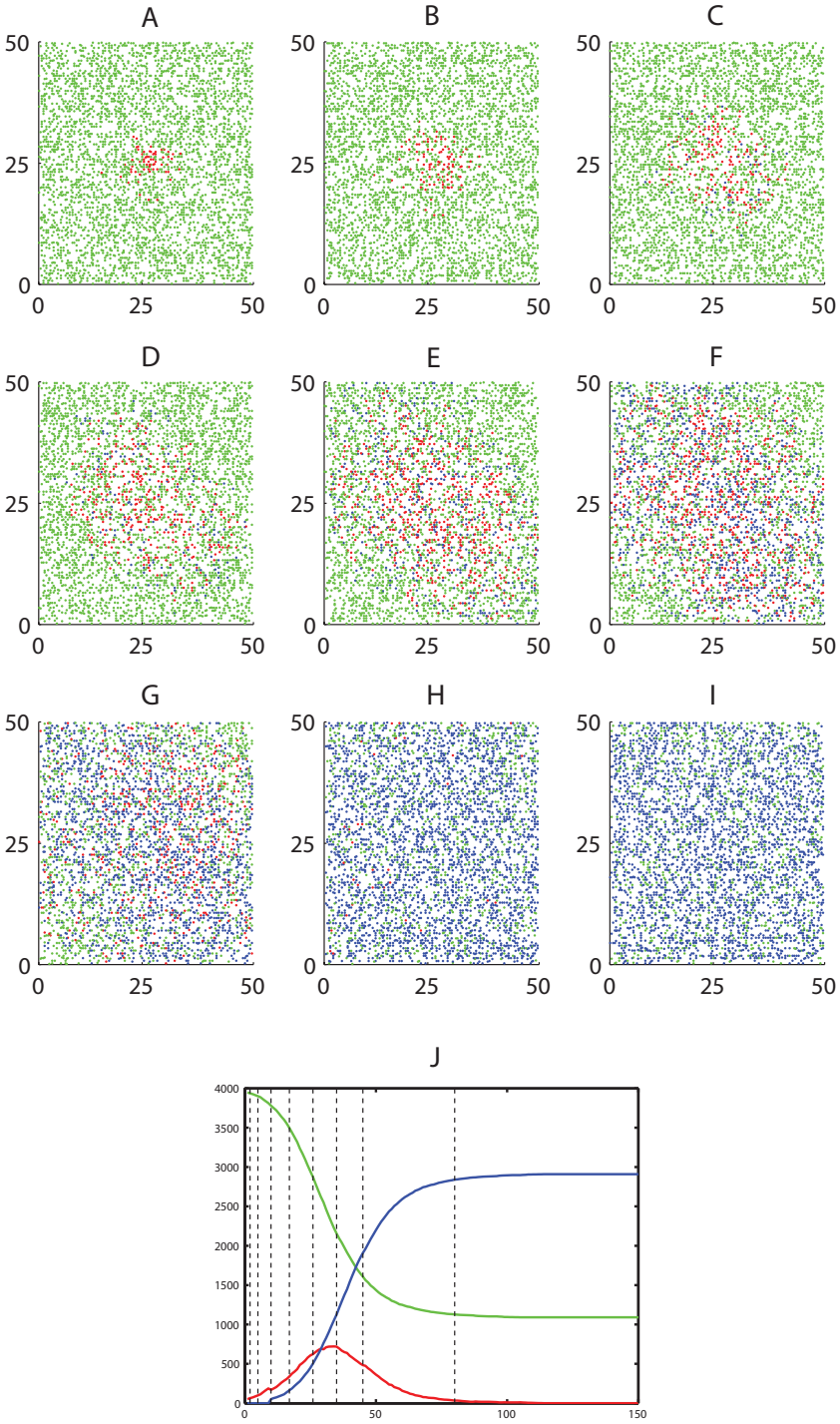

Figure 3.12: Visualization of epidemic spread of infected agents starting agglomerated in the middle of the cellular grid at begin of the simulation and cellular density of 0.26. Time steps: 1(A), 5(B), 10(C), 17(D), 26(E), 35(F), 45(G), 80(H), 150(I)

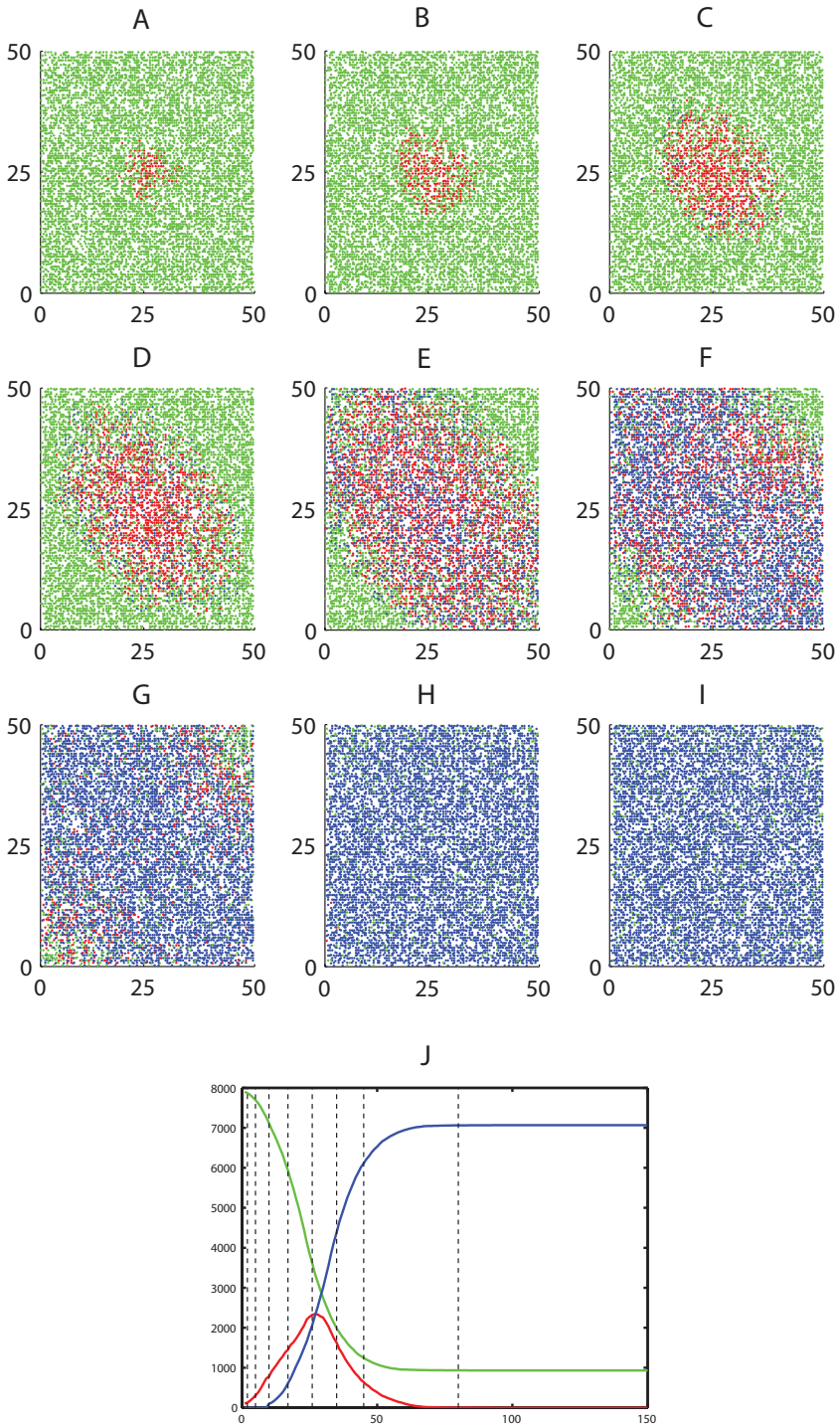

Figure 3.13: Visualization of epidemic spread of infected agents starting agglomerated in the middle of the cellular grid at begin of the simulation. Density is corresponding to the basic simulation parameters (0.53). Time steps: 1(A), 5(B), 10(C), 17(D), 26(E), 35(F), 45(G), 80(H), 150(I)

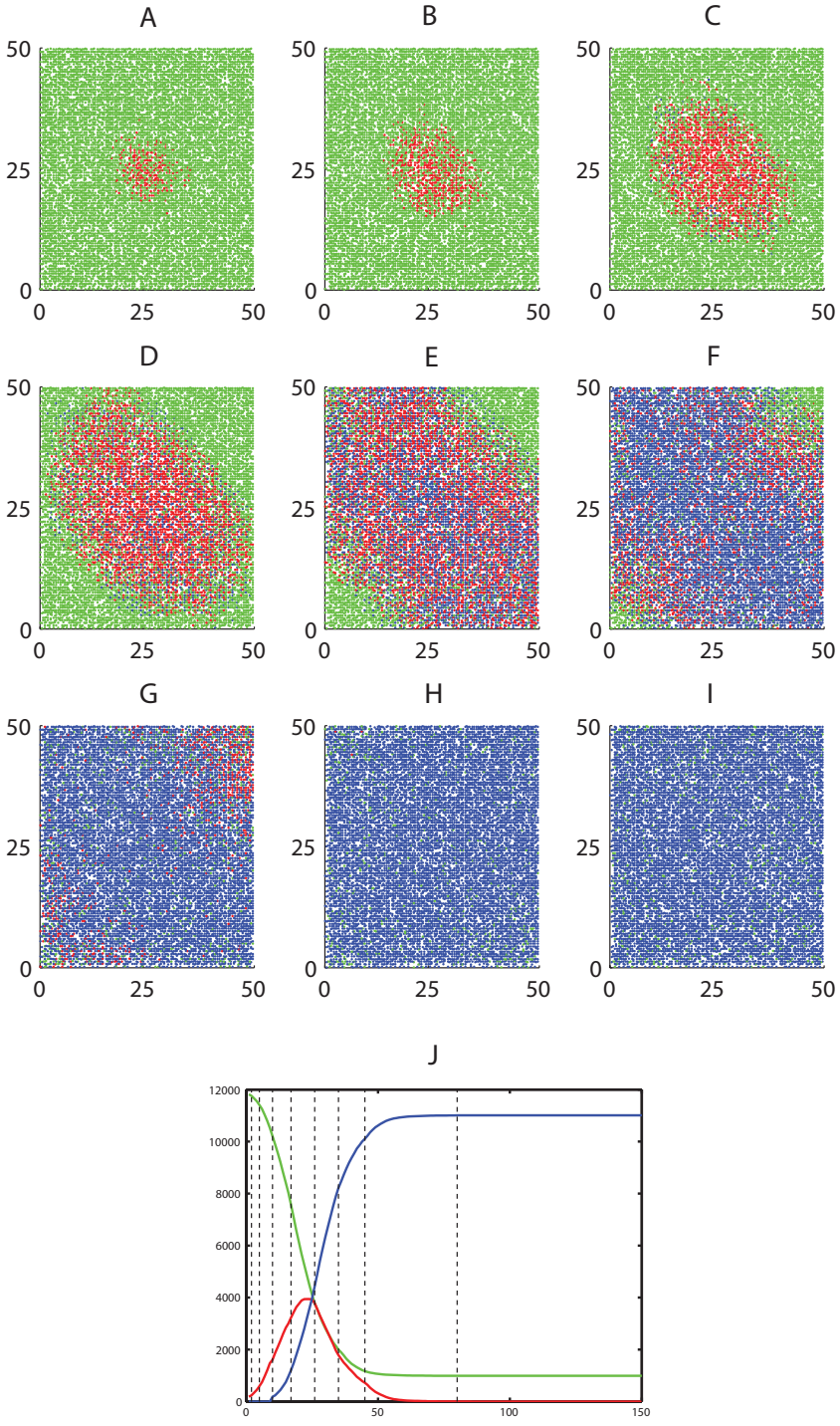

Figure 3.14: Visualization of epidemic spread of infected agents starting agglomerated in the middle of the cellular grid at begin of the simulation and cellular density of 0.8. Time steps: 1(A), 5(B), 10(C), 17(D), 26(E), 35(F), 45(G), 80(H), 150(I)

tacts. Nevertheless, in the same way as the CA-based model, the computing time of the random contact model is discrete.

The random contact model uses the agent based methodology to provide individual agents with attributes as the health status that can change during the simulation time. Similar to the previous model, these individual agents are connected to epidemic calculations of the SIR model. If a contact between an infectious and susceptible agent happens, an infection of the susceptible individual is plausible due to a defined infection probability. When an agent changes its health state to “infected" on the other hand, he remains infectious for other agents for a certain period of time (*infection duration*) and changes his health status to “recovered" when the remaining duration reaches zero.

Contacts between agents are generated in following way; First, a contact rate k has to be defined. The contact rate is the averaged amount of contacts an agent can have during one simulation of defined discrete time duration. When k is defined, agents are being assigned to each other by the random generator with respect to certain requirements. First, an agent is not allowed to have a contact with itself. Second, if every contact between two agents increments a contact counter by two (see details in the next chapter), the total amount of contacts to achieve in the random generation per time step is $k * n/(2)$ (where n is the amount of agents in the simulation). The assigning of contacts can be implemented by creating a container including all agents at the beginning of every time step which will be further used to regulate the set of agents that are available for random draw.

The contact rate k can be set to any number that should represent the averaged amount of contact per agent during the epidemic simulation, but because we want to provide a comparison between the random contact model and ca-based model in the next chapter, we try to find an estimator for the contact behaviour of the ca-based model. Before we do this, we have to define an important assumption for the comparison. As it can be seen in the previous chapter, different distributions of agent on the cellular grid (e.g. clusters) and modulations of starting position of infected agents have a significant effect on the results of an epidemic spread simulated with the CA-based model. The most important assumption is therefore that all particles inside the cellular grid are uniformly distributed, as it is defined in the basic configuration of the CA-based model (chapter 3.2). In the next step we have to set the contact rate parameter k of both models equally but when we look at the definition of the CA-based model we can see that this parameter does not exist and we have to estimate k from the dynamics of the FHP cellular automaton.

Given a hexagonal grid with horizontal and vertical size of dim_x times dim_y and N uniformly distributed agents, the probability p of an agent occupying one of six lattice vectors can be described as

$$p = \frac{N}{6 * dim_x * dim_y} \tag{3.3}$$

From the probability p we can deduce the probability q_i of a cell to be occupied by $1 \leq i \leq 6$ agents by

$$q_i = \binom{6}{i} p^i (1-p)^{(6-i)}, 1 \leq i \leq 6 \tag{3.4}$$

Having the probability q_i we can estimate the average amount of epidemic contacts per cell K as

$$K = \sum_{i=2}^{6} \binom{i}{2} q_i \tag{3.5}$$

Finally an estimator for the contact rate of the LGCA k_{ca} can be computed by dividing the estimated total amount of contacts with the amount of agents, described as

$$k_{ca} = \frac{K * 2 * dim_x * dim_y}{N} \tag{3.6}$$

The results of the comparison of the cellular automaton based model with the random contact model configured with the contact rate k_{ca} will be presented in the next chapter.

3.6 Comparison between CA-based model and random contact model

After the CA based model was presented in chapter 3.2 and the random contact model has been explained in chapter 3.5, it is interesting to compare both models in terms of generated results at equal starting parameters to identify cases where results differ or no comparisons can be made.

As it has been described in the previous chapter, the contact rate k of the random contact model has been set to the estimated contact rate k_{ca} of the ca-based model. We have to bear in mind that we can only compare the random contact model with ca-based model if particles are assumed to be homogeneously distributed at the whole grid during the entire simulation. Under this assumption, we compute the amount of *average contacts per agent and time step* during simulations with basic parameter settings (defined at chapter 2.3) and varying densities. Doing this, we record all occurred contacts between agents during simulations of both models with a contact counter and use it to calculate the average number depending on the length of the simulation and amount of agents in the observed grid. It has to be noticed that a generated contact between 2 to n agents at any time step increases the contact counter by the value c calculated as

$$c = \binom{n}{2}; 2 \leq n \leq 6 \tag{3.7}$$

Table 3.1 shows the obtained *average contacts per agent and time step*, that have been simulated by both models with equally set basic simulation parameters and varying grid densities.

As it can be seen at Table 3.1, the comparison of the amount of *average contacts* shows that under the configuration of the identified contact rate k_{ca} the random contact model indeed computes very similar results as the ca-based model (max. deviation = 3.4%).

Proceeding in the comparison of the models, we look at the development of epidemic spread computed by the random contact model at Figures 3.15 to 3.17 which can be compared to the results of cellular based model simulation at Figures 3.2 to 3.4 of chapter 3.3. Figure 3.15 shows the computational results of the random contact model with *basic simulation parameters*

Grid density	CA-based model	Estimated number of contacts
0.8	1.9951	1.9976
0.66	1.6670	1.7013
0.53	1.3337	1.3795
0.4	1.0011	1.0010
0.26	0.6683	0.6685

Table 3.1: Average contacts per agent and time step

when the amount of simulated agents is decreasing or increasing, resulting in a different cellular density ρ. Figure 3.16 shows the simulation of varying infection probabilities at equal basic simulation parameters (0.5%, 1%, 2%, 3%, 4%) while Figure 3.17 shows the model behavior at change of the infection duration of an agent (3 to 16 time steps).

In comparison with the previously presented results of the CA-based model we can view at following details; As the numbers at Table 3.1 suggest, the amount of *average contacts per agent and time step* generated by the random contact model configured with the contact rate k_{ca} is slightly higher than at simulation results of the CA-based model. This effect can be observed at comparison of subplots A to C of Figures 3.15 and 3.2 in particular at the faster rise of the epidemic population at random contact model results. While the CA-based simulation at lowest density (subplot A) reaches the peak of appropriately 1200 infected agents at the 25th time step, the random contact model generated 1500 infected individuals 5 time steps earlier. However, the CA-based model generates a comparable total amount of infected agents at a shallower time course so the total amount of infected, susceptible and recovered agents at the end of the simulation is very similar. A comperable development can be observed by at other cases of Figures 3.15 and 3.2.

Simulations of varying infection probabilities show an even higher degree of similarity at presented SIR development plots at Figures 3.16 and 3.3. While the generated amount of the infected population reach a slightly higher peak at the results of the random contact model, the development leads to a comparable amount of agents at every health status at the end of the simulation. Similar results are shown in the comparison of infection duration at Figures 3.17 to 3.4, where subplots B-E present minimal deviations of the SIR development and subplot A show a slightly different result of susceptible and recovered agents due to faster rise of inefctious contacts at the beginning of the simulation.

All in all the comparison suggests, that in simulation cases which are independent on the spatial structure of the grid or the containing agents, a replacement of the cellular automaton model by the random computation model will lead to similar results of contact determination if the contact rate parameter k has been set to the probability k_{ca} properly. On the other hand, it has to be mentioned that it is impossible to compute inhomogeneous distributions of agents or varying spatial morphologies with the random contact model, for such simulation problems (e.g. simulation of vaccination barriers) the use of the CA-based model would be a more adequate choice.

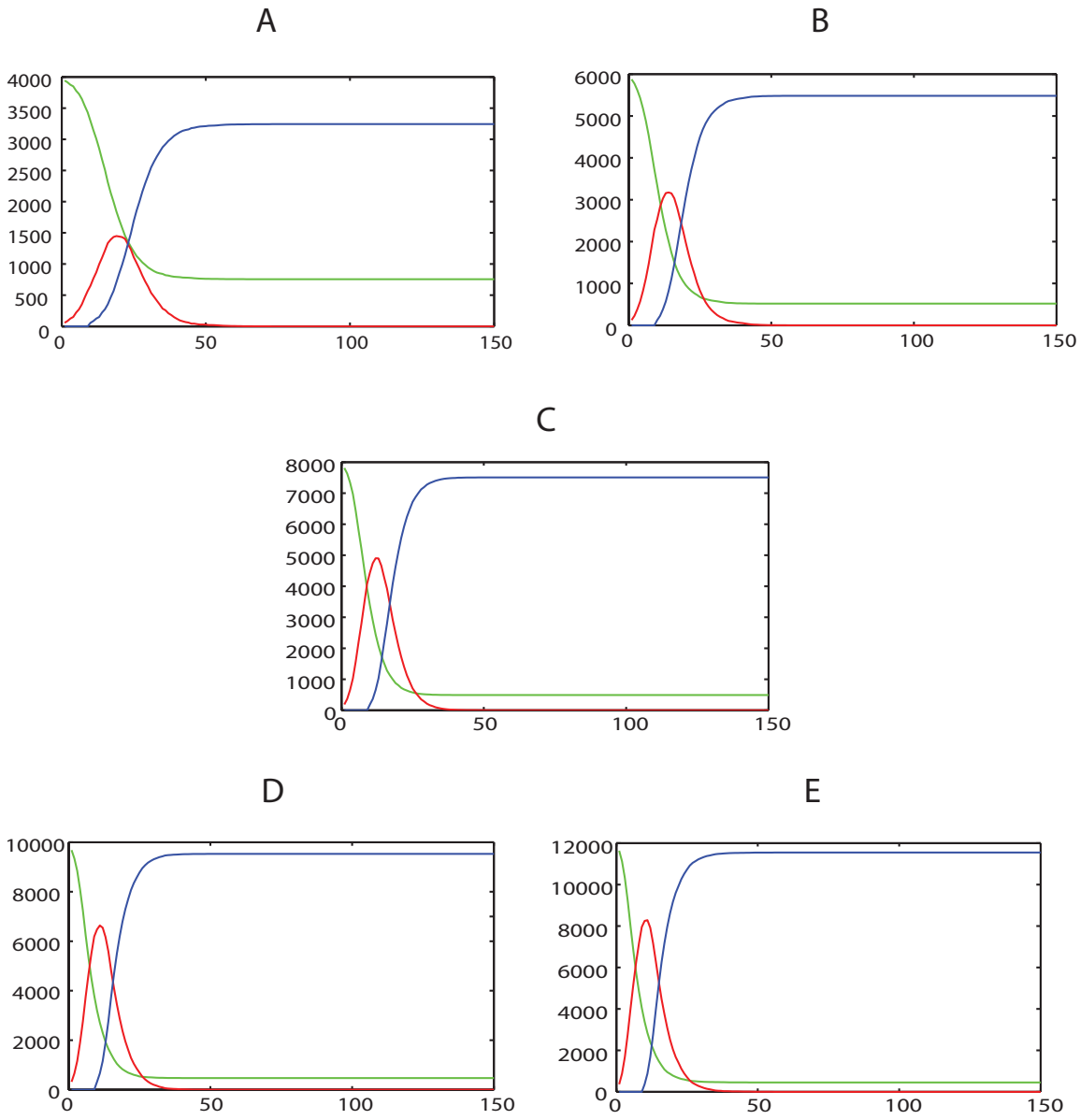

Figure 3.15: Epidemic spread computed with the random contact model. The amount of available agents is modulated so the values of grid densities (although the random contact model does not use a grid) would be 0.26 (A), 0.4 (B), Basic simulation run (C), 0.66 (D) and 0.8 (E)

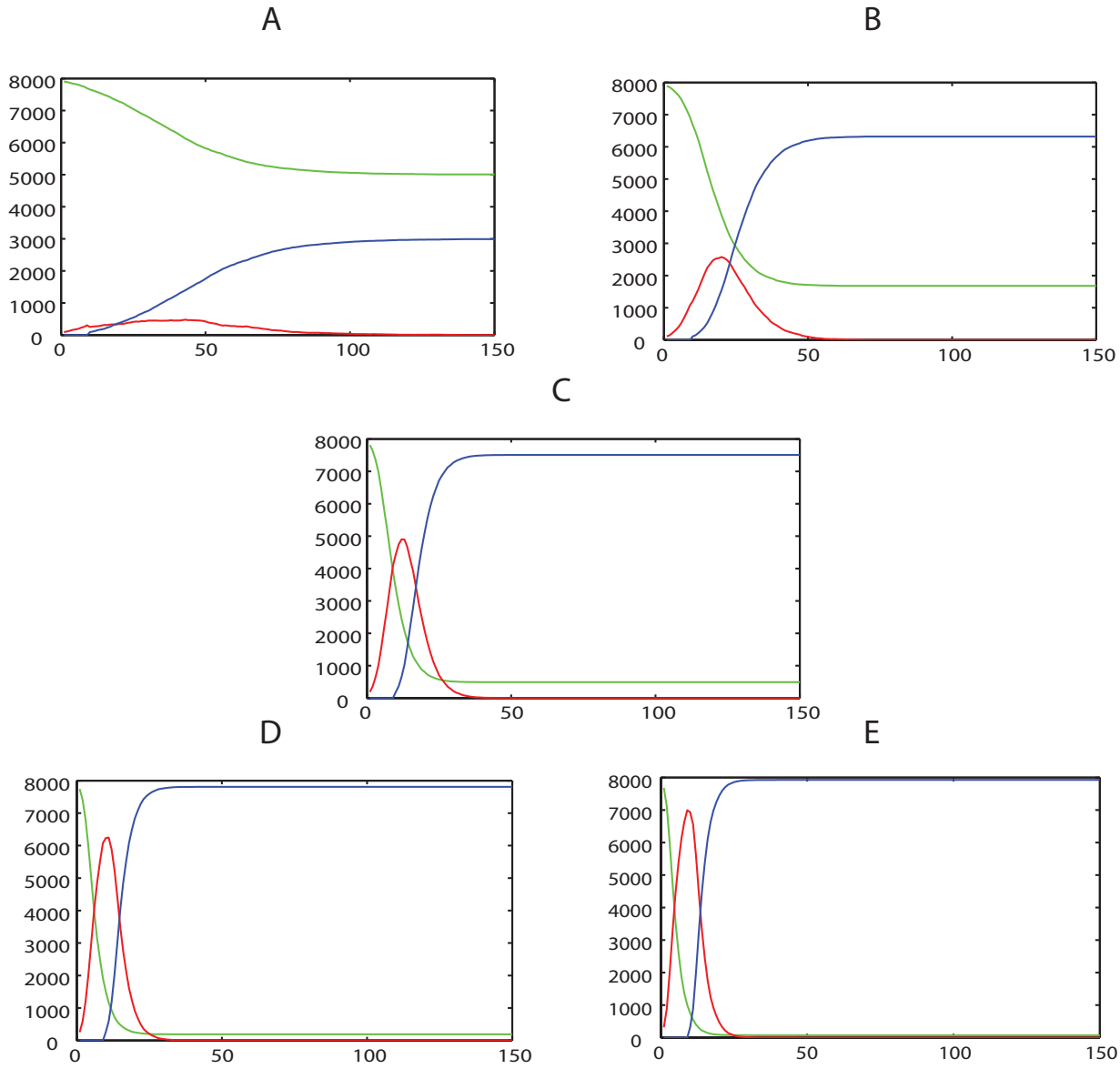

Figure 3.16: Computation of the random contact model with basic simulation parameters and varying infection probability to 0.5% (A), 1% (B), Basic simulation run (C), 3% (D), 4% (E)

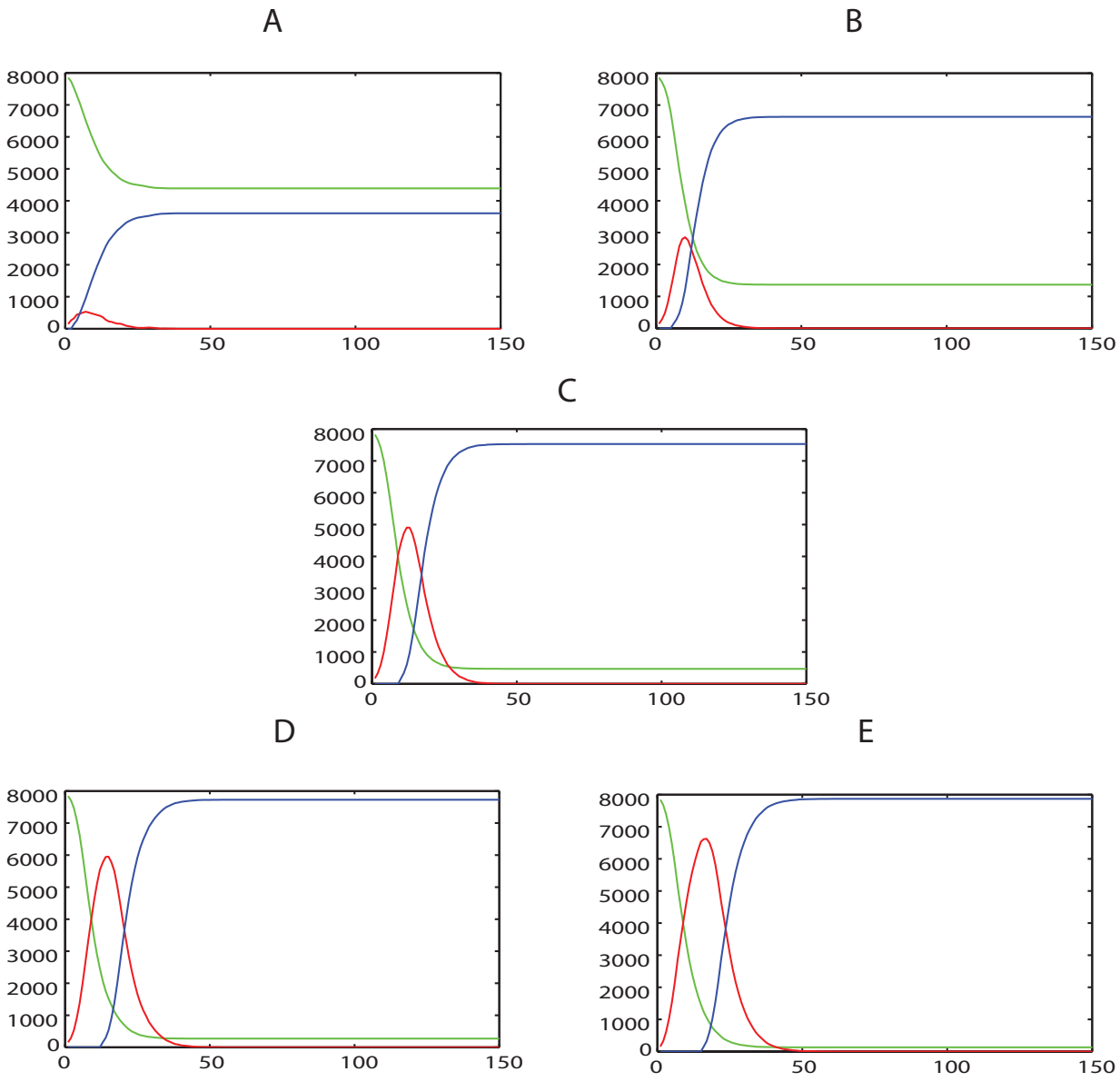

Figure 3.17: Computation of the random contact model with basic simulation parameters and varying infection duration to 3 time steps (A), 6 time steps (B), Basic simulation run (C), 13 time steps (D) and 16 time steps (E)

CHAPTER 4

Parallel Implementation

The aim of this chapter is to present an parallel implementation of a FHP cellular automaton on the Graphics Computing Unit (GPU) architecture. First a short overview will be provided on previous approaches in this area. Before presenting the actual parallel implementation a short overview on GPUs, their application in general purpose computing and the used programming enviroment Computer Unified Design Architecture (CUDA) in combination with MATLAB will be given.

4.1 Previous approaches

As stated in section Section 2.3, the invention of FHP automata in the year 1986 and the proof of their yield to Navier-Stokes equations aroused a high demand of solving complex problems in the field of computational physics and chemistry and the need to port the FHP LGCA algorithm to a parallel architecture with a high computational performance. Theoretical models of the deterministic CA proposed that due to the similarity of the algorithm to a basic concept of parallel data processing - the Universal Turing Machine - the development of a parallel method should be feasible [19]. Nevertheless, in comparison to basic 2D cellular automata the parallel implementation of the FHP LGCA has to deal with higher computational complexity due to the introduction of non-deterministic collision rules which decide if particles rotate left or right at an occurrence of a collision.

In the year 1987 Hayot et al [18] presented a first FHP LGCA high performance implementation using an INTEL iPSC parallel computer equipped with 32 16bit-x86 processor nodes arranged in an ethernet-connected hypercube with an absolute computing performance of 25 Mflop/s. The computation time of one update of a 4096x2048 cellular lattice reached a minimum of 3.5 seconds, involving the FHP-I set of movement rules expressed in 74 binary operations. The problem of non-determinism however, could not been solved in terms of providing fully probabilistic collision rules as it has been proposed by Fischler et al. Because the implementation of a parallel pseudo-random number generator has shown to be very performance

consuming on the given architecture, the authors decided to obligatory shift particles to the the clockwise direction of a node, whenever a collision occurs.

Figure 4.1: The CAM-8 [Source: http://www.ai.mit.edu/projects/im/cam8/]

In 1993, the CAM-8 cellular automata machine (Figure 4.1) has been developed at the MIT Laboratory for Computer Science, including a hardware circuited data stream for movement and collision handling directly in the system architecture [20]. Each site of the lattice had a certain number of bits (a multiple of 16), referred as a “cell", that could be translated through the lattice in any arbitrary direction. The CAM-8 parallel supercomputer evolved a D-dimensional cellular space with 32 million sites where each site had 16 bits of data with a site update rate of 200 million per second.
There is no information about the generation of random numbers on the CAM-8, but a remark on an implementation at a similar computer architecture, the CM-5. The non-deterministic collision direction there was derived by following function: if a collisions happened at even time step $(mod 2 = 0)$, the particles were shifted to the clockwise direction and if collisions happened on odd time steps $(mod 2 = 1)$, the particles were shifted to the counter-clockwise direction [20]. This function does not fully implement the theory of the FHP-I collision rule at a mathematical sound level, but proposes a non-monotone and mostly efficient computation of a two-state choice at a limited per-site memory of a parallel system architecture.

One decade later, the interest in FHP LGCA supercomputer simulations in the field of computational physics and fluid dynamics decreased due to advent of high-performance cluster computing and realizations of more accurate PDE solvers and numeric methods (e.g. finite elements). Nevertheless general applications of CA have risen in other fields, as for example image processing. More recent implementations were done on IC-architectures or Field Programmable Gate Arrays (FPGAs) [24], as well as GPU-systems [21] [23] [22] as it became known that consumer graphic cards could be used to implement parallel computations on a small scale. Depending on the type of implementation and memory handling, it has been demonstrated to possibly achieve a 55x-400x reduction of computing time of basic 2D cellular automata (with square grid size

and deterministic movement rules), when comparing the execution of an equal cellular grid on a consumer GPU card and a sequential implementation on the CPU.

4.2 Function and Design of GPUs

To most readers, the Graphics Computing Unit is mainly known from its usage in their PC systems, typically in the rendering of graphics and providing system output on our monitors. It is generally known that the efficiency at manipulation of computer graphics relies in the design of the highly parallel architecture of GPUs, which makes them more effective in processing small, geometric operations on a high amount of data in a parallel way.

This data processing paradigm is called *Single Instruction Multiple Threads* (SIMD), that is basically another name for *Single Instruction Multiple Data* (SIMD) of Flinn's computing architecture classification. In contrast to the CPU (which is classified as *Single Instruction Single Data*), the GPU performs the same instruction set simultaneously on a high amount of data. As it will be explained later, the execution hierarchy of the GPU can recruit up to thousands of parallel threads that are able to fractionize a computing problem into small pieces, perform the computation, and reassemble them to a global result.

Furthermore, the design of a CPU is optimized for sequential code performance, so threads are generally processed in sequential order - waiting for termination of one process before starting an other. The GPU on the other hand takes advantage of a large number of available threads and tries to schedule sleeping and pending threads dynamically so they are not blocking the execution of threads that are ready to run. In other words the scheduler of the GPU tries to minimize the amount of waiting threads and maximize the amount of running ones [26].

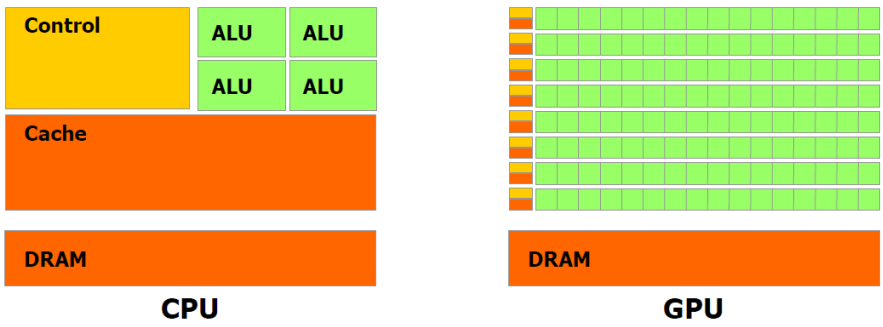

Figure 4.2: The streaming processor of Nvidia G80 chip in comparison with a general CPU architecture. The amount of arithmetic units (ALU) is much greater on the GPU, whereas the Cache and Control unit is less important than on CPU architecture [28].

The architecture of modern GPUs is organized into an array of so called *streaming multiprocessors* (SMs) which in turn consist of a number of *streaming processors* (SPs). The streaming processors share control logic and instruction cache (that is much simpler and lighter than on a

CPU as it is shown in Figure 4.3) and shared resources for all SPs - for example *shared memory* and *registers* (Figure 4.3).

The widest storage of the GPU is the DRAM, also often referred to as *global memory*. In graphical and video applications it is used as frame buffer memory or storage of texture information because it can store the majority of data on the GPU. Although the DRAM is constructed as a very-high bandwidth memory, the access latency to data is high - a process executed at one of SMs needs in average 460 clock cycles to access information at a given DRAM address [32]. Other memory resources on the GPU (e.g. shared memory and registers) provide a much lower access latency but their memory size is very limited so their use must be exactly calculated.

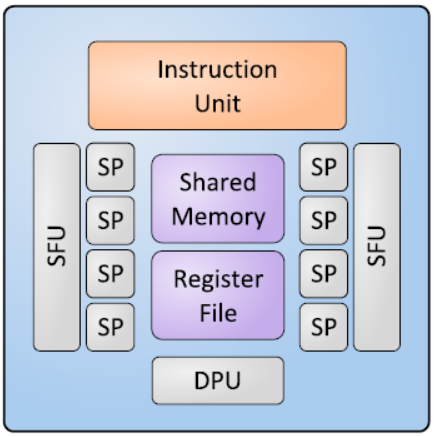

Figure 4.3: The streaming processor of a Nvidia G80 chip [32]

Further system units on streaming multiprocessor of the GPU include Special Function Units (SFU) which are responsible for executing mathematical functions such as root or cosine and the Double Precision Unit (DPU) that handles computations on 64-bit floating operands, if this precision is requested by the programmer [32].

4.3 General Purpose GPU Computing

In the year 2010, nearly all industrial available computers have been shipped with built-in GPUs and multi-core CPUs, shoving a growing trend towards parallelization in the consumer sector [26]. The performance increase of the GPUs is shaped by the fast growing video game industry, which puts a lot of effort to perform implement a high number of floating-point calculations per video frame in modern games. [25]

Due to the rising parallel performance for an affordable price, it was just a question of time until graphic programmers started to experiment with general purpose computing on GPU architectures. General Purpose GPU Computing (GPGPU) is no invention at its own, but rather

a summarization of methods and technologies that use GPUs not to render graphics and animations but to carry out these usual (CPU-like) computations on the device.
The first approaches of “misuse" were done in graphical programming languages such as *OpenGL* or *C for Graphics*. Since a few years however, the industry reacted to the rising attention in the GPGPU sector and started to develop programming environments specially customized for GPGPU. One of these proprietary environments, Computer Unified Device Architecture (CUDA), will be presented in the following chapter. In the last years of the GPGPU development it has been shown that the parallel GPU architecture can be used to compute complex scientific problems that could be solved only by huge and expensive supercomputers in the past. It is remarkable, that in the year 2010 a GPU (e.g. NVIDIA GeForce 580) has a hundred times higher theoretical throughput (1,581 Gflops) [28], than a CPU with one core (e.g. Pentium 4 with 14 Giga-Flops), but for a comparable price per chip. Of course the purpose of the GPU and CPU differs - the GPU is not designed for execution of complex instruction logic and control of multiple I/O data so the role of a CPU in nowadays PC systems can’t be overtaken by a GPU. Nevertheless, industry trends [31] show that these two architectures may unify towards an one-chip architecture that could enable a faster communication and sharing of sequential and parallel computing tasks between them.

4.4 Computer Unified Device Architecture

The Compute Unified Device Architecture (CUDA), introduced by Nvidia company in 2007, is a parallel computing architecture designed to simplify the development of parallel applications on Nvidia GPUs. it offers a software interface to most common programming languages as C and Fortran and provides an access to GPU structures for general purpose computing, as for example; memory handling, execution control or synchronization of threads. On the next pages, a basic overview of this architecture will be provided to explain its function, advantages and limitations;

Execution hierarchy and scheduling

As can be seen in Figure 4.4, the execution of threads with CUDA is organized in a hierarchical order; *grids of blocks of warps of threads* [28]. This dimensioning of the hierarchy helps the GPU to map a desired amount of threads onto hardware resources in order to manage efficient execution of programs.

The executed programs on the GPU devices are called *kernel functions*. In general, all running threads of a GPU are processing one kernel function at a given time, so every thread is computing the same copy of kernel sequence on data from a distinct source and destination address in the memory.

The developer can set two levels of the hierarchy when designing a kernel: number of blocks (or “Grid Size") and number of threads (“Block Size"). In regular cases, the least number of thread to chose is the number of necessary elements of the result matrix. As it will be described in detail in chapter 4.5, in case of the LGCA this number equals the amount of cells of the cellular automaton. Furthermore threads need to be defined by unique coordinates to distinguish

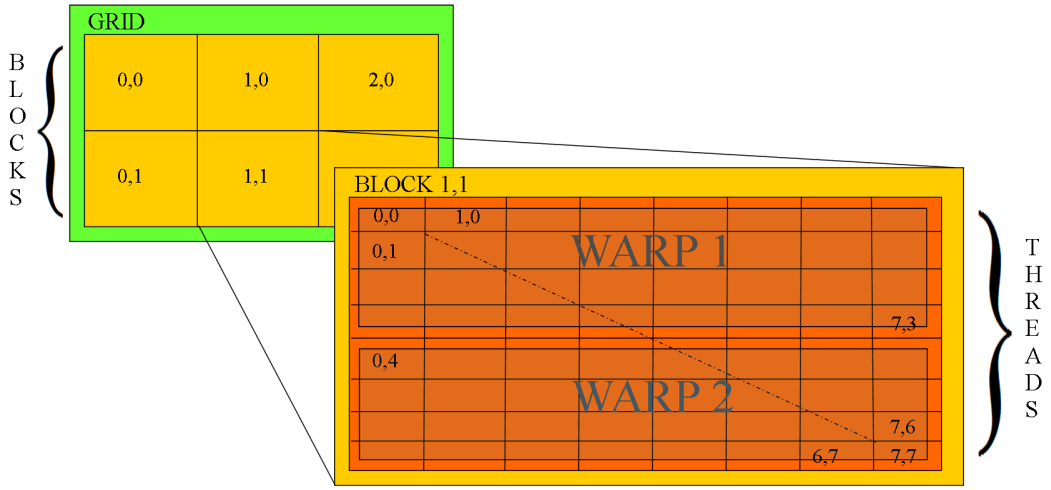

Figure 4.4: The CUDA execution hierarchy: Grids of blocks, blocks of warps and the summation of 32 threads to one warp

themselves from each other and to identify their position in the memory used by the function. This can be done by using a two-dimensional thread index, which in our case will be equal to X and Y coordinates of cellular automaton cells. The programming sequence necessary for the use of threads coordinates will be discussed in the next chapter.

Figure 4.4 shows the arrangement of threads into warps. Each warp consists of 32 threads of sequenced index values. In fact, warps are not part of the CUDA specification; however because they are the the unit of thread scheduling in SMs, knowledge of them can be helpful in optimizing the performance of parallel applications. When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution, but a earlier available warp is chosen. This mechanism is known as *latency hiding* [26]. For the CUDA developer, it is an important fact to know; when one thread of a executed kernel function goes to pending state, all other 30 threads from the same warp will be scheduled for a delayed execution as well.

Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads and assigns them to one SP of the GPU [26]. The runtime tries to keep all threads of a block at one SP, so they can use the same shared resources and synchronize each other. The prerequisite is that enough resources are available for the desired amount of threads.

Memory model

CUDA supports several types of memory. As it is presented in Figure 4.5 they do not distinguish only by their size or type, but also as by the way they can be accessed from programs running on the CPU or GPU. For example, the *global memory*, *constant memory* and *texture memory* can be

read and written by the host (which is the where the execution of every CUDA application starts - the CPU) and by all running threads of the GPU, even if they belong to different blocks. Other memory types, which are surrounded by yellow squares at Figure 4.5, can not be read or written by the host. Only the GPU itself can access this areas, so if we want to transfer data to one of them, we have to pass it over through one of the host-writable memories. Furthermore, registers are assigned only to *one specific thread* whereas shared memory access is limited to *all threads of one block*.

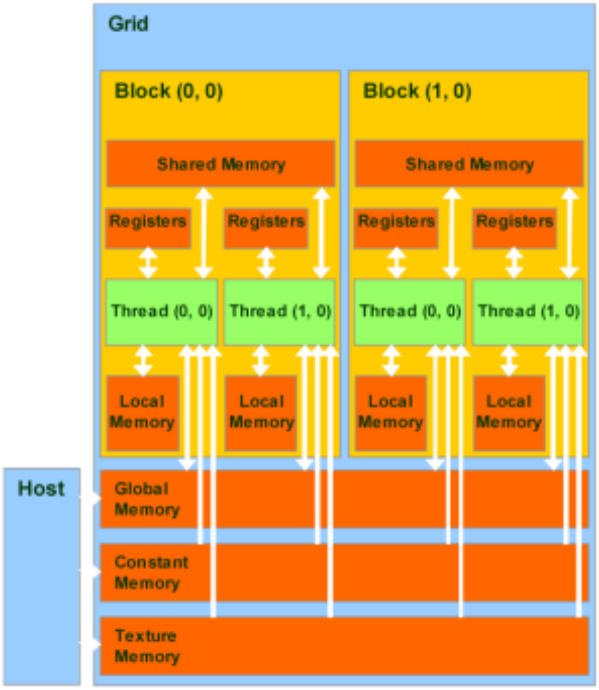

Figure 4.5: CUDA memory model [28]

To present further attributes of GPU memory ressource, we specialize on the *GT200* GPU architecture, that has been used for development of the LGCA algorithm and is very similar to most of Nvidia manufactured GPUs at the time of writing. We divide the memories of the GPU into DRAM-located and on-chip memories and start with enumeration of the *DRAM-located* resources:

- *Global memory* is the widest and most frequently used memory located in the DRAM. Its access is uncached and has a documented latency of 400-600 cycles [28].

- *Texture memory* is a cached, read-only memory space. It has a slightly lower access latency that the global memory due to the cache when performing 2D and 3D addressing [32].

- *Constant memory* is a short-latency (8 clock cycles [32]), high-bandwidth memory which works in cached modus when all threads simultaneously access the same location, but it is restricted to read-only access for all GPU threads [26].

As can be seen in Figure 4.3, *on-chip-memories* reside next to streaming processors, therefore they are the fastest accessible resources in a parallel application.

- *Registers* can be accessed four times per clock cycle by every SP, so they provide sufficient bandwidth to execute three read and one-write operand instructions (e.g., multiply-add) at every clock cycle [32]. Registers are reserved to individual threads, so each thread can only access its own registers. Because of that, kernel function typically use registers to store frequently used variables that are private to each thread.

- *Shared memory* is available to all threads of one block only, as can be seen in Figure 4.5, it is therefore used for block-intern coordination and sharing of short-latency data. The amount of shared memory allowed per block is 16 KB, at the GT200. The kernels function parameters also occupy the shared memory, thus they slightly reduce the usable memory size [26].

The advantages and disadvantages at use of presented memory locations can be summarized in a following way; The global memory is the widest available storage on the GPU, but it has the highest read and write access latency. Nevertheless, it is necessary to communicate with the host and share data between itself and the the threads of the GPU. A way to speed up access of the global memory is the so called *coalesced read and write method*. When addressing a half-warp (16 words) of consecutive memory locations in a kernel execution, the access to the memory banks will be united into one conjoint operation. Because the support for coalesced operations differs on every generation of Nvidia GPUs, it is best to consult the actual CUDA guide [29] for details of its usage at the given architecture. The suitability of texture memory and constant memory usage is often limited due to tricky addressing of the texture memory and the read-only access of the constant memory on GPU kernels. Registers and shared memory resources are fast, but very restricted in size. For example, if there are 512 threads to run on a SM, only 30kb of shared memory space is available per thread - that is barely enough to store of 14 integer or float variables during execution of the kernel. If more variables are needed, the CUDA scheduler automatically recruits regions of DRAM-located memory space for their storage - in the most cases, this reduces the total performance of the CUDA program dramatically.

Synchronization and Control Flow

The most important point to know about execution of parallel threads on GPU devices is that they are *asynchronous*. If the host-program decides to execute a kernel, it will pass all necessary data and parameters to the CUDA scheduler, which recruits a certain amount of threads to run the desired kernel. From this point the host program can not decide how many threads will be running at which certain time and in which exact order. It will be notified when the total amount blocks and threads finished their computation the result can be transfered back from the GPU memory.

Because CUDA offers no settings on thread execution timing, the only way to synchronize threads that are depending on each other is the *barrier synchronization function*. When a kernel function calls the sequence *syncthreads()*, the thread that executes the function call will be held at the calling location until every other thread in the block reaches the same location [28]. This ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase [26].

It has to be mentioned, that the barrier synchronization function should be use with caution, because repetitive waiting for threads reduces the overall performance, as if we imagine one group completing a task that has to wait for the slowest part of the group before it can start another task. This can be described as the major tradeoff in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other because none of them must wait for each other.

Another phenomenon that can cause significant decrement of the overall program performance is *branching* due to diverse execution paths of a warp. Any flow control instruction (if, switch, do, for, while) can cause threads of the same warp (32 threads) to diverge. Examples could be, if one part of threads of a warp execute the positive path of an if-instruction and another part the negative one or some threads need to run a for-loop more often than another ones. If this happens, the different execution paths on the GPU must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path [29]. In the practical use, it can be therefore more efficient to run all paths of a program without divergence operations and decide a question at post processing of the data on the CPU.

4.5 Using CUDA in connection with MATLAB

Since the release of Parallel Computing Toolbox v 5.0 it is possible to use *MATLAB* for writing CUDA compatible host code that can be used to setup kernel properties such as thread or memory dimensioning and execute pre-compiled GPU kernels directly from the *MATLAB* environment. This chapter should be used as an introduction to CUDA in connection with MATLAB with examples that are direct provided from the source code of the acctual implementation. For more detailed instructions on developing CUDA applications please consider the CUDA programming guide [28] and the MATLAB Parallel Toolbox Documentation [27].

1. Compiling CUDA Code
   The source code of a CUDA application is usually written in text-based source files, which can be compiled to runable parallel thread execution binary (PTX) with the NVCC compiler. This compiler can be obtained free of charge from the Nvidia website and can be used to generate binaries in the system console as the following example shows:

   ```
   nvcc -ptx lgca_agents_shared.cu
   ```

2. Setting up a connection with the kernel
   Using MATLAB and the parallel computing toolbox, we can create a *CUDAKernel* object that is virtually linked to the CUDA source code file (*.cu) and the PTX-binary (*.ptx) and will be later used to trigger the execution of the kernel. The last argument provided is the name of the entry function in the kernel, in our case “runLGCA".

   ```
   kernel = parallel.gpu.CUDAKernel('lgca_agents_shared.ptx',
   'lgca_agents_shared.cu','runLGCA');
   ```

3. Execution Configuration
   As discussed in chapter 4.4, we have to dimension the amount of running threads with respect to the CUDA execution hierarchy. The highest level of the hierarchy is the grid, that is composed of a two dimensional matrix of blocks. The second level of the hierarchy is the block, consisting of a two dimensional matrix of threads. These both parameters can be altered with the *GridSize* and *ThreadBlockSize* property of the *CUDAkernel* object. In this example, we have chosen a block size of $20 * 20$, resulting in 400 threads per block and a grid size of $2 * 2$ which leads to a total allocation of 1600 threads (Because every thread will process one cell, this equals to 1600 cells from a $40 * 40$ cellular automaton grid).

   ```
   kernel.GridSize = [2 2];
   kernel.ThreadBlockSize = [20 20];
   ```

4. Memory transfer to GPU
   In the starting state of a LGCA, there cellular grid can be set to different spatial configurations of agents. Therefore, it is more flexible to create a grid configuration on the CPU and copy this data as a input onto the global memory of the GPU. In the Parallel Toolbox environment, these input matrices need to be casted into a data type called *GPUArray*. By executing the commands shown above, data will be transfered from the workspace of *MATLAB* to the global memory of the GPU. One must bear in mind that unlike on CPU programs, the amount of global memory is limited to the size of DRAM of the GPU. If one oversteps this size of the global memory, the transfer will be canceled.

```
Db1=parallel.gpu.GPUArray(caGrid{1});
Db2=parallel.gpu.GPUArray(caGrid{2});
Db3=parallel.gpu.GPUArray(caGrid{3});
Db4=parallel.gpu.GPUArray(caGrid{4});
Db5=parallel.gpu.GPUArray(caGrid{5});
Db6=parallel.gpu.GPUArray(caGrid{6});
```

5. Kernel execution
   At this step, the input data is prepared and the kernel is ready for execution. The Parallel Processing Toolbox offers the function *feval()* to execute the kernel and define the input and output data to be process. The results of the kernel will be lead back to the return values indicated at the left hand side of the command. Be aware, that after calling *feval()*, the results still reside on the memory of the GPU and have to be copied to CPU with the command *gather()*. In the case of the LGCA algorithm, two further integer parameters have been passed to the GPU (for one-dimensional parameters, the use of GPUArray is not necessary) which represent the width and height of the grid.

```
[Db1,Db2,Db3,Db4,Db5,Db6]=feval(kernel,Db1,Db2,Db3,Db4,Db5,Db6,m,n);
```

6. Memory transfer to CPU
   After execution, the result can be copied into the workspace of *MATLAB*. As presented below, the function *gather()* takes a *GPUArray* object that resides on the memory of the GPU and moves it into a matrix of the same size on the CPU. In the case of the LGCA implementation, every lattice vector dimension has its own two-dimensional matrix describing the occupation of cells. On the GPU, the matrices are represented as six different variables, on the CPU they are saved in a *MATLAB* cell array.

```
caGrid{1}=gather(Db1);
caGrid{2}=gather(Db2);
caGrid{3}=gather(Db3);
caGrid{4}=gather(Db4);
caGrid{5}=gather(Db5);
caGrid{6}=gather(Db6);
```

The Parallel Computing Toolbox offers a suitable way to handle GPU computations inside MATLAB and include calls to CUDA kernels in greater MATLAB projects. On the other hand, when compared to the native environment of CUDA in C the possibilities to configure CUDA calls involve just a basic set of instructions and no possibility to change device memory data types or execution parameters. Also it has to be pointed out, that the Parallel Computing Toolbox supports only newer GPU devices, to be exact from Compute Capability 1.3. If one wants to perform GPGPU computations with MATLAB on older cards, there is still another possibility to compile MEX files with CUDA [30], however it is not as handy as the presented solution.

4.6 Implementation Overview

In this chapter, the algorithm of the actual approach will be explained and an overview will be given about the most fundamental steps of the parallel implementation. The execution of the kernel in time is pictured schematically by the flow chart diagram at Figure 4.6. The figure shows memory storage of the cellular grid at course of time and indicates transfers from global to shared memory and contrariwise. Additionally, it pictures the main processing steps of the LGCA kernel, which will by explained in more detail with presentation of the underlying source code on the next pages.

1. Entry function and its parameters
   As it is explained in chapter 4.7, the hexagonal structure of the LGCA cellular lattice can be transformed to six regular 2-d matrices where each of them store one direction of particle vectors for every cell of the grid. These input matrices are stored as linear arrays in the global memory and are accessible through pointers *b1, b2, b3, b4, b5* and *b6*. The main routine of the LGCA algorithm starts with the following lines:

   ```
   __global__ void runLGCA (unsigned int* b1, unsigned int* b2,
   unsigned int* b3,unsigned int* b4, unsigned int* b5,
   unsigned int* b6, int m, int n)
   ```

2. Assigning shared memory and copying cells from global memory
   In order to increase the memory performance of the computation, parts of the cellular grid are transfered to shared memory where all relevant operations perform more efficiently. This technique is explained in more detail in the following chapter. Because the *BlockSize* (amount of threads per block) has been set to $20x20$ in the initialization of our application (see chapter 4.5), the size of the shared memory window has to be dimensioned equally. If this was done properly, every thread transfers one cell from the global memory to an assigned location in the shared memory plus the threads that are responsible for the lateral cells of the grid have to transfer additional *border cells* (see next chapter for details). All in all the size of the shared memory arrays *as,bs,cs,ds,es* and *fs* has to be set to $22 * 22$ cells, which will in sum occupy 11.34 kb (that is under the limit of 16 kb) of the shared storage.

   The assignment of shared memory can be written in *CUDA C* code as

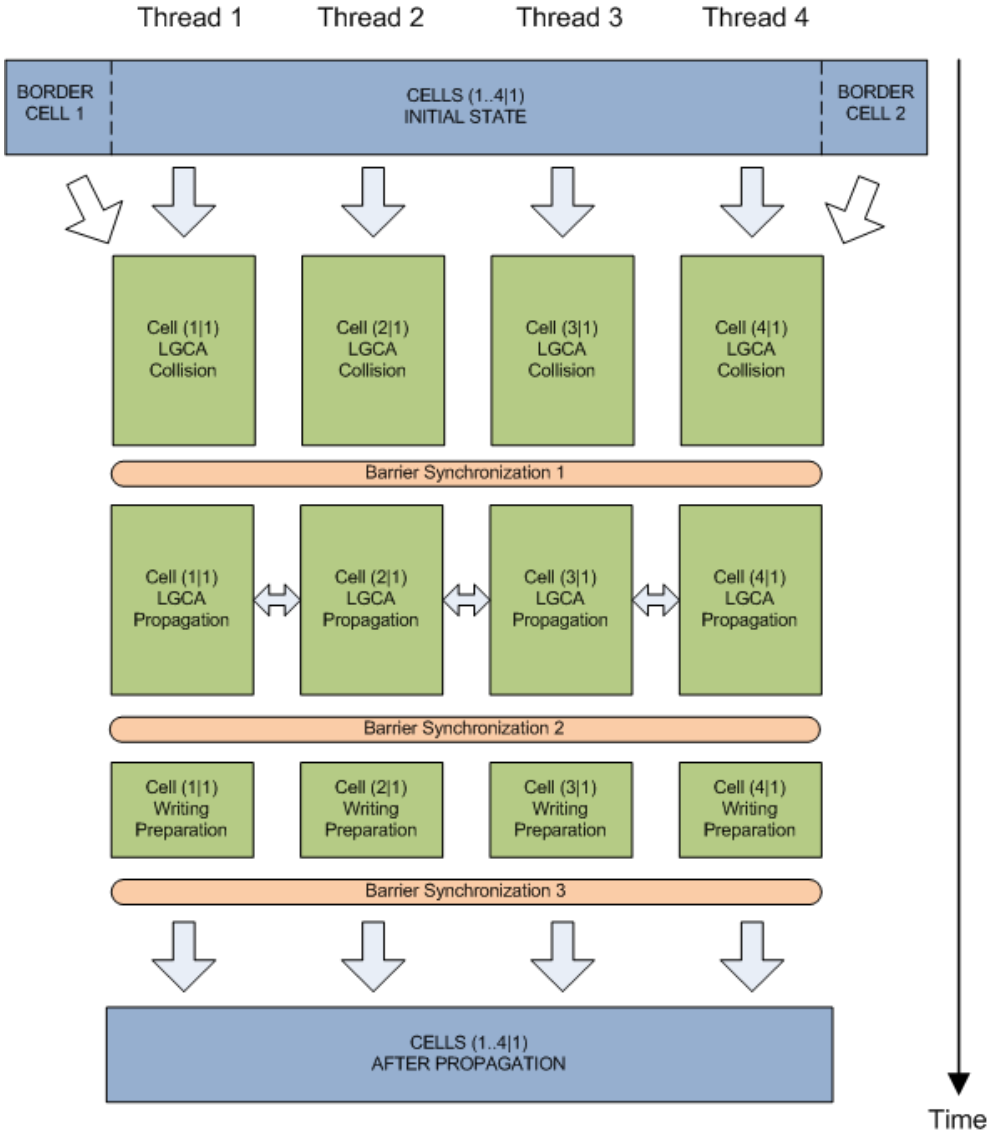

Figure 4.6: Schematic drawing of one time step in the parallel implementation of the LGCA in CUDA. The blue color indicates data stored in the global memory of the GPU, the green color indicates data stored in the shared memory. Each cell is processed by one thread, it is assumed that all threads belong to the same block (and have the same shared memory assigned). Blue arrows indicate transfer of cell data from global to shared memory (above), shared to shared memory (middle) or shared to global memory (below), white arrows indicate the transfer of border cell data to the most left and right threads.

```
#define SHARED_DIM_X  22
#define SHARED_DIM_Y  22

__shared__ unsigned int as[(SHARED_DIM_X)*(SHARED_DIM_Y)];
:
__shared__ unsigned int fs[(SHARED_DIM_X)*(SHARED_DIM_Y)];
```

where the colon stands for variables *bs,cs,ds,es* which are initialized with equal instructions.

Afterwards, a corresponding cell can be copied from the adress *linidxG* in the global memory to the adress *linidxS* in the shared memory by executing

```
as[linidxS] = b1[linidxG];
:
fs[linidxS] = b6[linidxG];
```

where the colon stands for equal instruction on shared memory variables *bs,cs,ds,es* and global memory variables *b2,b3,b4,b5*.

Additional border steps are copied, if the thread corresponds to the first or last row or column coordinates of the shared memory window. The underlaying algorithm of this method will be explained in detail in the next chapter.

```
if ( threadIdx.x == 0 && threadIdx.y != blockDim.y - 1 )
{
  fs[getAdr(0,tidy,  SHARED_DIM_X,SHARED_DIM_Y)]
  =b6[getAdr(x-1,y,  m,n)];
  es[getAdr(0,tidy+1,SHARED_DIM_X,SHARED_DIM_Y)]
  =b5[getAdr(x-1,y+1,m,n)];
}
```

The code from above is responsible for copying one particular region of “border cells" into the shared memory, in this case the left cell from the actual grid position. As it is explained in chapter 4.7, all lateral threads of a block have to execute similar commands to load boarder cells into their shared memory window.

Function *getAdr()* computes the linear address of a designated cell contained in either in the global memory grid or shared memory window.The arguments of function are x/y coordinates of the cell and the size of the grid in the memory. Because the function respects boundary overflow it is of practical use at many memory instruction in the kernel.

3. Particle collisions
   If two- or three-particle collisions are detected chiral rotations have to be computed upon these cells to maintain the dynamics of the FHP LGCA as presented in chapter 2.3. In our

implementation, the direction of the chiral rotation is decided by a modulo division resulting from addition of x and y thread coordinates which results in divergent execution of threads responsible for clockwise or counter-clockwise direction. Details on the collision handling are provided in the next chapter.

```
bool direction = (x+y)%2;
if(direction)
{
  temp=fs[linidxS];
  fs[linidxS]=es[linidxS];
  :
  bs[linidxS]=as[linidxS];
  as[linidxS]=temp;
}
else
{
  temp=as[linidxS];
  as[linidxS]=bs[linidxS];
  :
  es[linidxS]=fs[linidxS];
  fs[linidxS]=temp;
}
```

where both colons stand for memory copying instructions of the same pattern on remaining shared memory variables, so in the upper example all variables are shifted counter clockwise from *as,bs,cs,ds,es,fs* to *fs,as,bs,cs,ds,es* and in the above example clockwise from *as,bs,cs,ds,es,fs* to *bs,cs,ds,es,fs,as* (with the help of temporary variable *temp*).

4. Particle propagation
   Particles in the LGCA model propagate in certain directions dependent on their position in a particle vector at a previous time step. In the presented implementation, every thread gathers information on incoming particles by addressing cell data from defined neighbor coordinates relative to his own position in the grid. When the address is resolved, incoming particles do not immediately overwrite the positions of the actual particle vector but they have to be stored in temporary variables to avoid data collisions at asynchronous memory access. The address of the incoming particle vector may be computed in respect to the actual cell coordinates (variables *tidx* and *tidy*) and providing of horizontal and vertical dimension sizes *SHARED_DIM_X* and *SHARED_DIM_Y*.

```
nextA=as[ getAdr(tidx,   tidy-1,   SHARED_DIM_X, SHARED_DIM_Y) ];
:
nextF=fs[ getAdr(tidx-1, tidy,     SHARED_DIM_X, SHARED_DIM_Y) ];
```

where the colon stands for instructions of the same pattern used for copying values from shared memory variables *bs,cs,ds,es* to the temporary register variables *nextB,nextC,nextD,nextE* (see point “Cell propagation" of chapter 4.7 for details on addressing).

5. Writing of shared memory into global memory
   After all threads of a block completed the particle propagation step, the altered grid in the shared memory is ready to be transfered back to the global memory of the GPU device. As it will be explained in the next item, a barrier synchronization is necessary to ensure every thread finished the propagation and the result is ready to be written.

   ```
   as[linidx] = nextA;
   __syncthreads();
   if (x < n && y < m) //is thread inside the border of the grid?
   {
   b1[linidxG] = as[linidxS];
   :
   }
   ```

   The same instruction is executed to copy shared memory variables *bs,cs,ds,es,fs* to positions in global memory indicated by pointers *b2,b3,b4,b5,b6*.

6. Use of Barrier Synchronization
   As it can be seen in Figure 4.6, the implementation needs three calls to the barrier synchronization function during the computation of every LGCA time step. The first synchronization is necessary because all threads of the block have copied a piece of cell data to the shared memory and determine the particle collision detection rule on it. Before the cell propagation can be initiated, it has to be ensured that every possible chiral rotation has been carried out and the results can be shared between neighbor threads for further processing. If this synchronization fails, the same particle could remain duplicated at to two different locations, because it could be taken from another thread as source of propagation before it has been rotated to another particle vector.

   The second barrier synchronization is necessary, because at time of propagation, all threads of the block do the same thing; they contact they neighbors and get their cell data to update their own cell. Problems may arise from this situation, because one thread could update his cell data faster than another thread and the overwritten result of time step $t1$ would be available to his neighbors instead of the correct cell state of time step $t0$. To avoid this, temporary variables have to save the new state of the cell, which will be used for overwrite the current cell status after calling the barrier synchronization and ensuring that every thread finished his propagation step.

   The third call to barrier synchronization has to happen just before the writing of the updated cell status to the global memory, to ensure that every thread finished the setting of its current cell status and all shared memory variables reside in correct state. After every thread exports the results back into the global memory, the complete cellular automaton grid has been updated by one discrete time step.

4.7 Implementation Details

Transformation of the hexagonal lattice

As shown at Figures 4.7 and 4.8, the hexagonal lattice can be directly transformed into a 2D matrix because it can be seen as a graphical adaptation of diagonal connectors shifted to left and right at alternating rows. Nevertheless, the neighborhood is the same as the 2D Von-Neumann definition (see chapter 2.3) with addition of two diagonal neighborhoods per cell, considering the translations of two particle vectors towards cells of the upper and lower row.

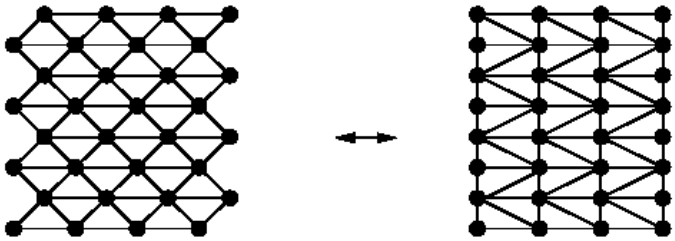

Figure 4.7: Mapping a hexagonal mesh onto a standard 2D array [33]

Figure 4.8: Transformation in detail

Loading cells into shared memory and creating border cells

To enable the use of high-bandwidth shared memory in the LGCA implementation, a method often described as *Tiling* in the literature [26] was used to increment overall efficiency of the algorithm. Tiling ca be imagined by a two-dimensional window of defined size, which is sliding over a bigger grid in both directions (Figure 4.9) and processing the data inside the boundary of the window. To be exact, this data will be transfered to the shared memory, processed by the kernel and copied back to the global memory when all instruction are executed. This method gains further performance from the use of the faster coalesced global memory access (see the CUDA memory model description in 4.4) because all threads processing data from the tiling

window have to access their source data from a sequential address in the global memory.

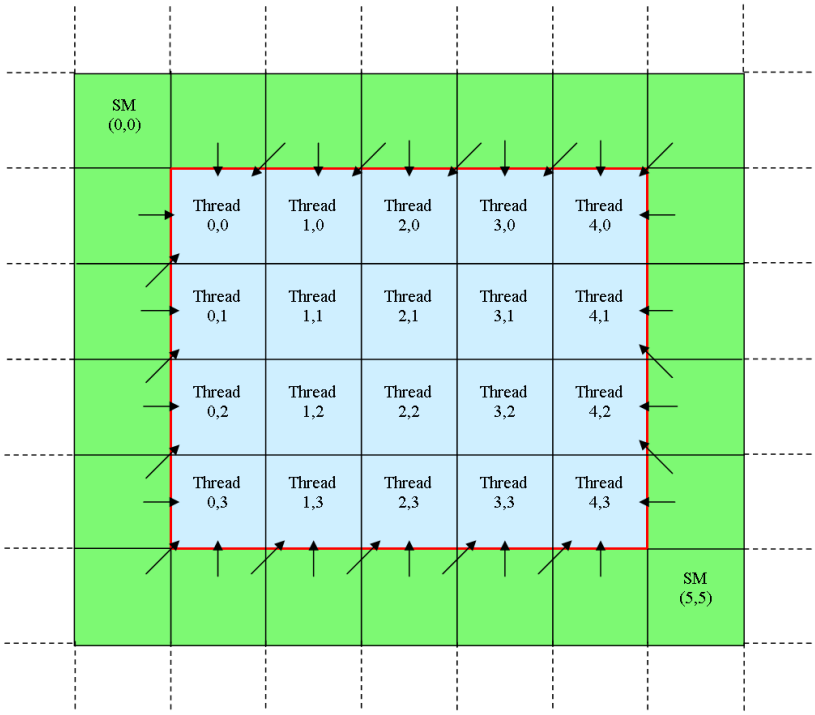

Figure 4.9: Shared memory windowing of all threads of one block and use of border cells (indicated with green color) at processing of the cellular grid. Black arrows indicate the propagation directions of particles of the border cells.

As shown in Figure 4.9, border cells (presented in green color) have to be additionally loaded into the shared memory by the threads at the lowest and highest x and y coordinate in every block. In this example, thread $(2, 3)$ has to load two border cells from locations below and diagonal to the left, so particles from these cells can propagate into the cell occupied by thread $(2, 3)$. Thread $(0, 3)$ has to load border cells from three memory location, because it is located on the corner of the window. This implies that three particles are going to potentially propagate from the border cells and three others from the cells occupied by threads $(0, 2)$, $(1, 2)$ and $(1, 3)$.

To construct a shared memory window with enough space for border cells, both dimensions have to be increased by two (As shown in chapter 4.6, the resulting dimension is our case is 22). When all contained particles are shifted by 1 in X and Y direction, the border cells can be stored at rows $(x, 0)$, $(0, y)$ and $(xdim + 2, 0)$, $(0, ydim + 2)$.

Collision rules

Collisions considered are either of the 2-body or 3-body type, due to the reasons of implementing FHP type I, as discussed in chapter 2.3.

- *3-body collisions.* At given lattice a 3-body collision responds to the following: if particles arrive from *a, c, e* directions (with no particles from *b, d, f* directions at the moment), they scatter into into *b, d, f* directions. In terms of the binary representation of lattice variables this consists in going from state (010101) to state (101010) or vice versa. The computation at each node consists of determining whether the current configuration is either of these two, and if so, switch to the other one. The computation for a set of threads can then be performed simultaneously at world level by setting following binary variables:

$$u_{3col} = ((a \wedge \neg b) \wedge (c \wedge \neg d) \wedge (e \wedge \neg f)) \vee ((b \wedge \neg a) \wedge (d \wedge \neg c) \wedge (f \wedge \neg e)) \quad (4.1)$$

- *2-body collisions.* Here the rules are following, there has to be decided which part of the particles is collided:

$$u_1 = (a \wedge d \wedge \neg(b \vee c \vee e \vee f)) \quad (4.2)$$

$$u_2 = (b \wedge e \wedge \neg(a \vee c \vee d \vee f)) \quad (4.3)$$

$$u_3 = (c \wedge f \wedge \neg(a \vee b \vee d \vee e)) \quad (4.4)$$

  Resulting in the global indicator

$$u_{2col} = (u_1 \vee u_2 \vee u_3) \quad (4.5)$$

  Depending on this boolean variable a chiral rotation may, or may not be initiated.

Cell propagation

The propagation of particles on the lattice at time step t is computed by resolving the neighborhood connection introduced in chapter 2.3. If written with help of two-dimensional coordinates, the propagation rules can be summarized as

$$P_1(t, x, y) = P_1(t - 1, x, y - 1) \quad (4.6)$$

$$P_2(t, x, y) = P_2(t - 1, x + 1, y - 1) \quad (4.7)$$

$$P_3(t, x, y) = P_3(t - 1, x + 1, y) \quad (4.8)$$

$$P_4(t,x,y) = P_4(t-1,x,y+1) \tag{4.9}$$

$$P_5(t,x,y) = P_5(t-1,x-1,y+1) \tag{4.10}$$

$$P_6(t,x,y) = P_6(t-1,x-1,y) \tag{4.11}$$

Where P_n is a two dimensional matrix storing one of $1 \leq n \leq 6$ particle vectors for all hexagonal cells which can be referred to with linear x/y coordinates. If x or y coordinates reach the grid boundaries at $x = \{0, ydim\}$ or $y = \{0, ydim\}$ the addressing points to overlapping coordinates at x or y direction, so for example $P_n(t, xdim+1, 0-1)$ leads to position $(t, 0, ydim)$ of the cellular grid.

Chiral Rotation

As discussed in chapter 4.1, when a parallel algorithm is about to be implemented on a massively parallel architecture, trade-offs have to be made in cases between computing complexity and performance in time. Because of the limited memory resources per thread (see chapter 4.2), it is hard to implement a sophisticated pseudo-random number generator beside a memory consuming algorithm without risking a obstacle in performance due to swap of variables into the global memory (that is about 55 times slower than the shared memory [32]). Because the direction choice of the chiral rotation relies on a 50:50 decision, it is not as critical as in other number-dependent applications to use simpler methods for computation, such as a modulo-division of thread-dependent variables as for example the x and y coordinates of the thread.

CHAPTER 5

Results of Parallel Implementation

In this chapter I will present the results of the computing complexity benchmark of the parallel implemented contact finding module (as a part of the CA-based epidemic mode, defined in chapter 3.2) and compare it to its functional equivalent implementation on the CPU. Furthermore a comparison between different rotation algorithms and their effect on GPU performance will be presented in chapter 5.2.

5.1 Comparison between CPU and GPU

In Figure 5.1 we compare the algorithmic performance in time of the CA-based contact finding module implementations on CPU, GPU and GPU with memory transfer to CPU after every simulation time step. The computing performance was measured with help of MATLAB function *tic* and *toc* called before and after the execution of the module. When we observe the benchmark of the GPU implementations we have to distinguish between two variants of execution to maintain a fair comparison between both architectures. When the module is executed on the CPU, results of computation are available ad hoc after every time step, so they can be used for visualization or detection of contacted agents. Unlike to the previous one this is not the case on the GPU because the cellular grid at time step $t+1$ (after the execution) remains on the memory of the device and can be not accessed directly from the CPU without the execution of an explicit memory transfer to the CPU. If this memory transfer to the CPU is executed only once - after the last time step - and the grid remains stored in the GPU during the whole simulation, we have to observe the computing complexity function indicated with red line in Figure 5.1. In case we are interested in the execution time in combination with repeated memory transfer after every time step of the simulation in order to analyze collisions of agents to compute SIR dynamics, we have to analyze time performance indicated by the green line in Figure 5.1.

The results propose a quadratic connection between time complexity of the simulations and an increase of grid dimension d, so that the increase is linearly proportional to the total amount of cells on the grid at the CPU and GPU implementation (as shown in Figure 5.2). While the

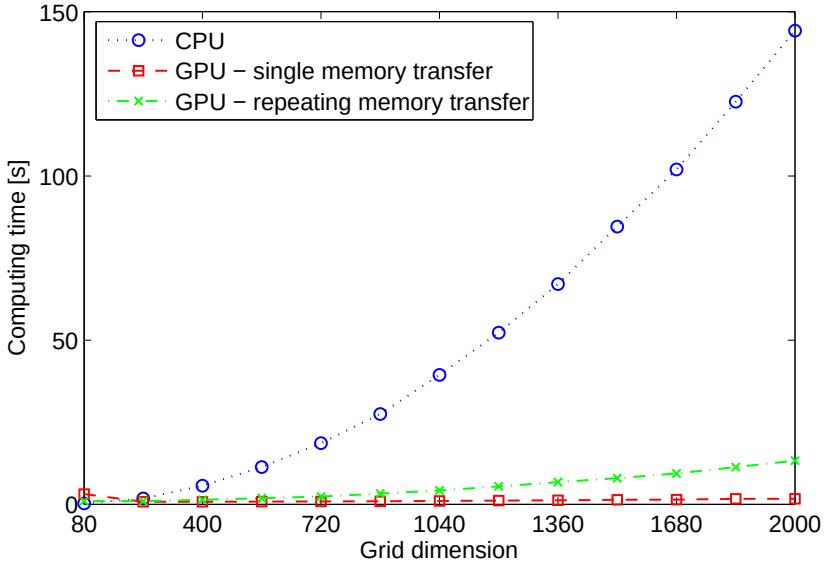

Figure 5.1: Comparison of the CA-module runtime on CPU and GPU with single or repeating memory transfer. The simulation was carried out at 100 time steps and an increasing cellular grid size of 80 to 2000 of vertical and horizontal dimension. Chiral rotation algorithm is used for collision handling in the GPU implementation.

CPU implementation performs better on small grid dimensions (see Figure 5.3) the performance of the GPU clearly predominates at increasing dimension sizes (Figures 5.1 and 5.2) or time steps per simulation (Figures 5.4 and 5.5). In numbers, the difference between CPU and GPU time consumption at simulations of 100 time steps and quadratic grid dimensions over 1000 cells are about 10 times higher when compared with execution with repeating memory transfer and 90 times higher when compared with execution with single memory transfer (at the initial and final state of the grid). The difference between the two variants of GPU executions may be explained due to the rising amount of data per grid ($4 * 10^6$ cells at the highest grid size) which has to be transfered after every execution. This bottleneck can be seen further at Figure 5.3, where the architectures are benchmarked at grid dimensions below 200. The initial data transfer to the GPU needs a constant minimal amount of time (approximately 0.7 seconds at our measurements), which is not possible to minimize regardless the size of the grid. As the CPU does not need to carry out any memory transfer before the execution and can access the grid directly through the *RAM*, it runs simulation of grids smaller then 140^2 in lesser computing time than the GPU. This relationship between computing costs expensed for memory transfer and the actual parallel computation is also well depicted at Figures 5.6 to 5.9. This analysis performed with *CUDA Profiler* shows the amount of spent GPU time on memory transfer from CPU to GPU (*memcpyHtoD*) or GPU to CPU (*memcpyDtoH*) as well as the processing of the grid (*runLGCA*). Apparently, at a dimension size of $d = 100$ and 1 time step per simulation,

the proportion of computation to memory transfer is 1 to 10. If we increase the amount of time steps per simulation to 100, the cost of memory transfer is decreasing to a negligible amount. When viewing the profiler result of simulating a cellular grid of dimension $d = 1000$ at 1 time step (Figure 5.8), it is visible how significantly the proportion of computing time to transfer time sinks at a huge amount of data in this simulation configuration. These effects are responsible to the high discrepancy between both GPU execution types at Figure 5.1. Last but not least, Figure 5.9 shows how this proportion may be minimized when the time steps are increased to 100.

The benchmark was performed on Intel Core2 CPU and Nvidia GeForce GTX465 GPU at a 16x PCIexpress slot, running on MATLAB in connection with CUDA as described in chapter 4.5. Collisions in the GPU implementation are handled with the chiral rotation algorithm (see next chapter for further details).

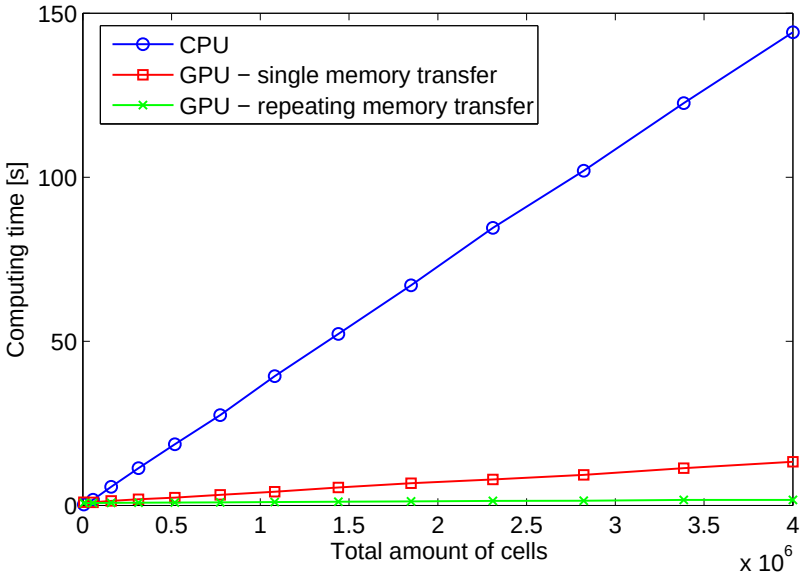

Figure 5.2: Comparison of the CA-module runtime on CPU and GPU with single or repeating memory transfer. The figure shows the linear connection between the total amount of cells and the computing time of the algorithms.

5.2 Comparison between different particle rotation algorithms

As written in chapter 4.7, there are several possibilities to implement particle rotations as part of the collision handling of the FHP-LGCA automaton on a parallel architecture as the GPU. In this comparison we take into account *chiral rotations* where 50 % of collided particles turn clockwise and 50% anti-clockwise as well as *monotone rotations* where all particles turn solely anti-clockwise and an implementation without any particle rotations. As stated in chapter 4.4 the

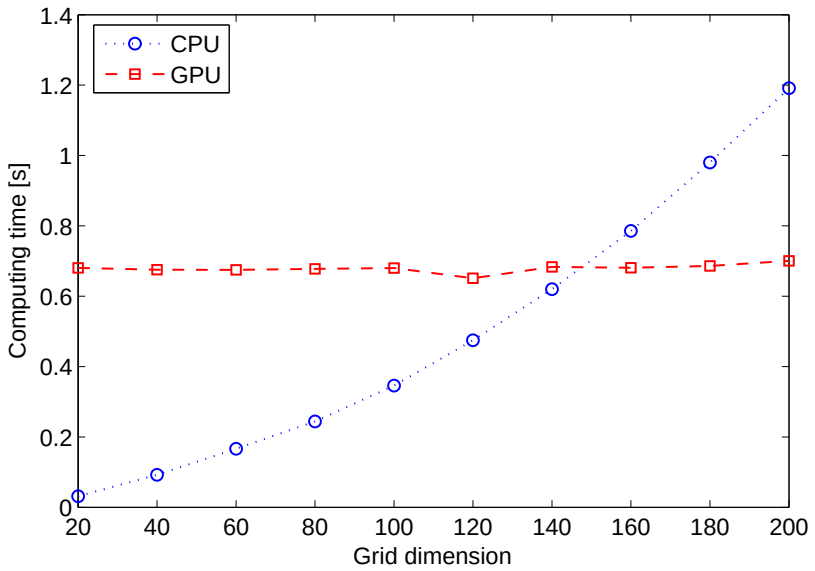

Figure 5.3: Computing performance of CPU and GPU (single memory transfer) compared at small grid dimensions. Due to the additional memory transfer to the device, the GPU can not use its parallel computing capability, while the CPU can access grid data directly and execute at lesser computing time.

resources on the GPU are limited and the computing performance may decrease with more advanced algorithms that use divergent execution paths. We compare simulations with a duration of 100 time steps and single memory transfer. As can be seen at Figure 5.10, the computing complexity functions of all implementations develop quadratically in proportion to the grid dimension and linearly in proportion to the amount of cells on the grid. Interestingly, implementations of monotone collisions are not consuming more computing time than implementations without collisions proposing a very small linear factor which is however not constant. The complexity function of the chiral rotation implementation has a steeper slope which indicates a higher linear factor than than the other possibilities. The main reason for these differences is that the implementation with chiral rotations bifurcates the execution paths in two parts: the threads processing the clockwise rotation and ones processing the anti-clockwise rotation. As the total amount of cells is rising with increasing grid size, more threads are needed for computation and due to the constant bifurcation the processing times is doubled with the amount of cells on the grid. This is not the case if collisions are handled with monotone rotations because these are executed sequentially with other parts of the LGCA algorithm.

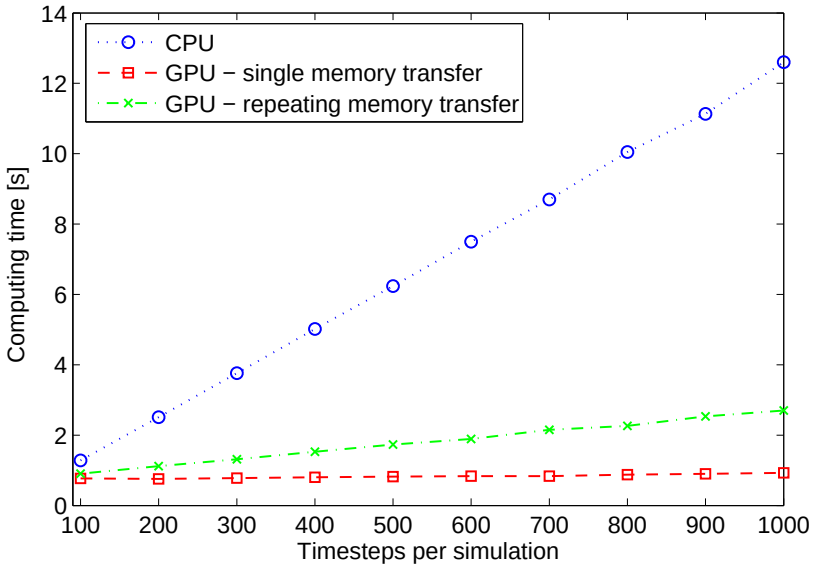

Figure 5.4: Comparison of the CA-module runtime on CPU and GPU with single or repeating memory transfer and increasing time steps per simulation on a lattice grid of $200x200$ cells.

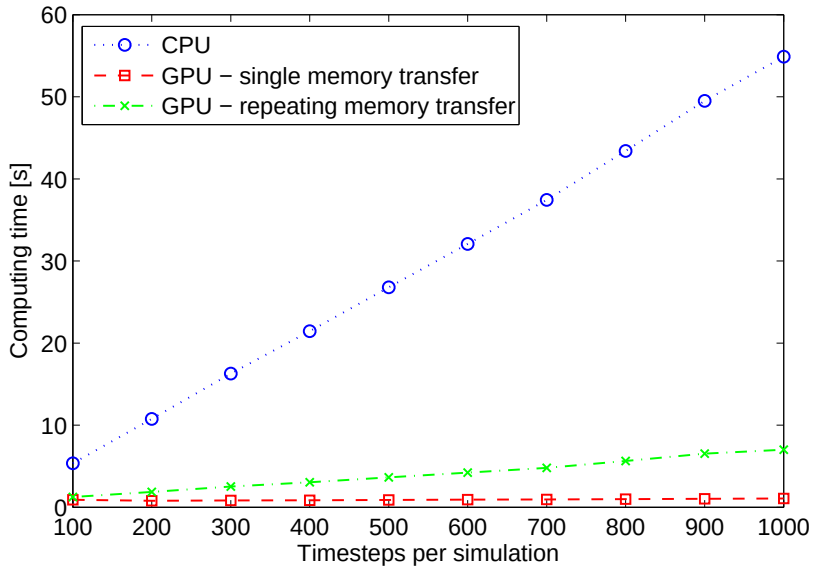

Figure 5.5: Comparison of the CA-module runtime on CPU and GPU with single or repeating memory transfer and increasing time steps per simulation on a lattice grid of $400x400$ cells.

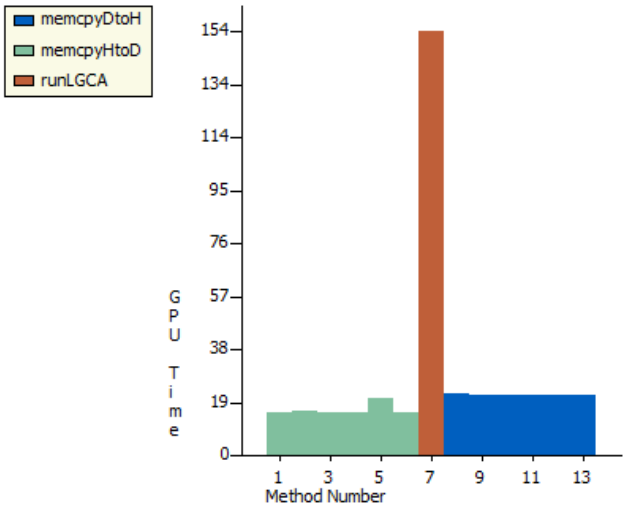

Figure 5.6: GPU Profiler analysis of processing a cellular grid of $d = 100$ at 1 time step.

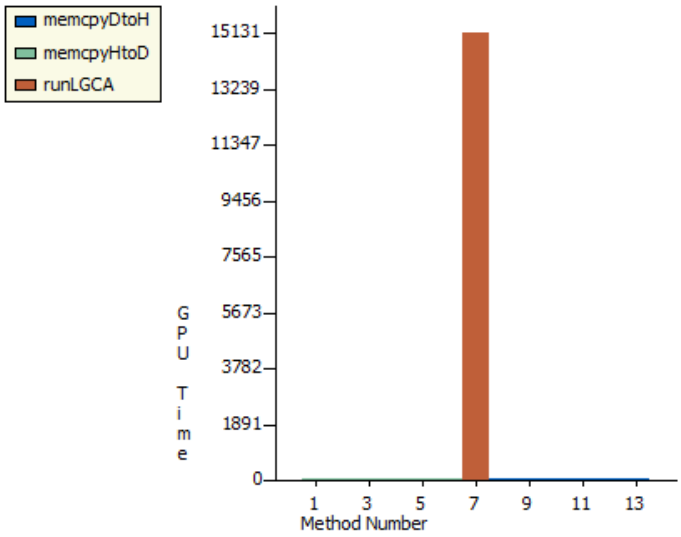

Figure 5.7: GPU Profiler analysis of processing a cellular grid of $d = 100$ at 100 time steps.

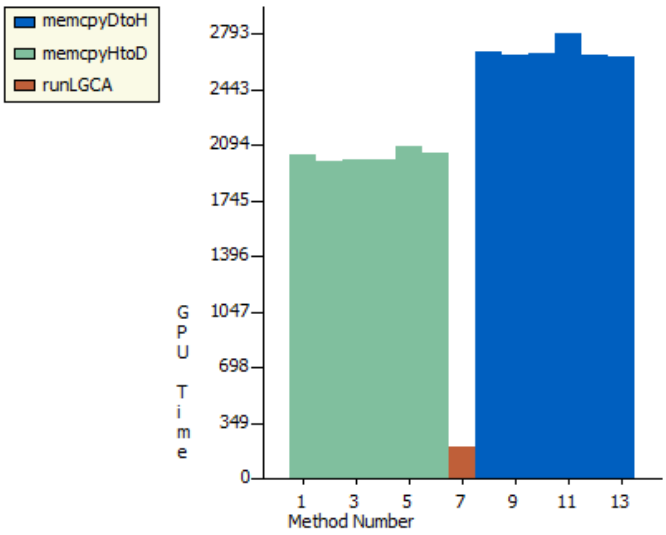

Figure 5.8: GPU Profiler analysis of processing a cellular grid of $d = 1000$ at 1 time step.

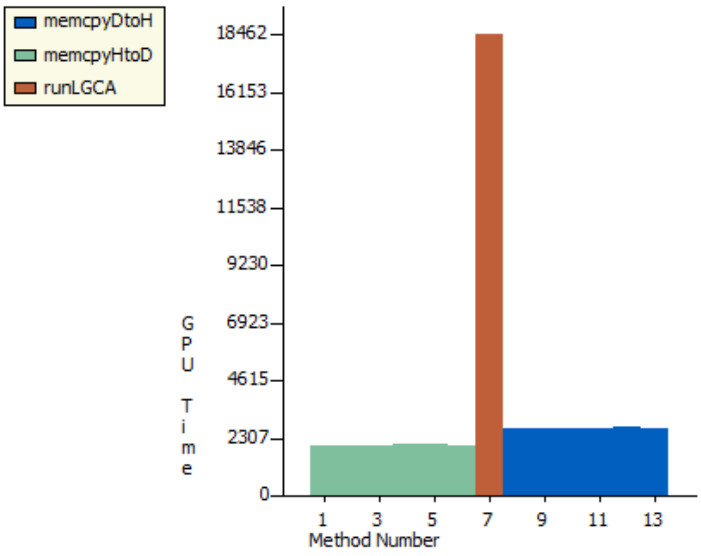

Figure 5.9: GPU Profiler analysis of processing a cellular grid of $d = 1000$ at 100 time steps.

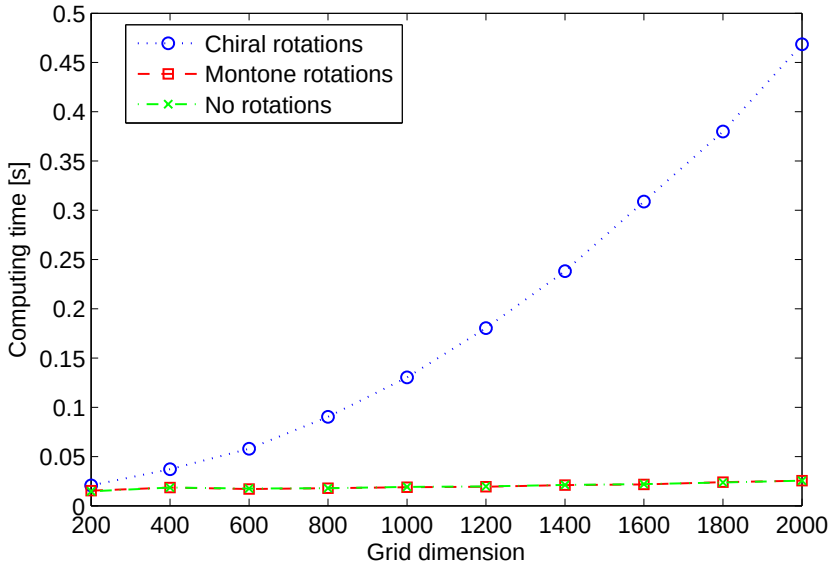

Figure 5.10: Comparison of GPU implementations with various way to handle chiral rotations at simulations of 100 time steps with increasing grid size. All implementations are compared in the single memory transfer mode.

CHAPTER 6

Conclusion

In this thesis I presented an epidemic model consisting of SIR dynamics (chapter 2.1), the agent based simulation method (chapter 2.2) and various implementations of the contact finding module which assigns agents to each other in order to induce an epidemic contact.

In chapters 3.2 to 3.4 we first dealt with a lattice gas cellular automaton (LGCA) based variant of the contact finding module which has been introduced in detail and visualized during various simulations of epidemic spreads on discrete space.

Thereafter, the random contact model (RCM) was presented in chapter 3.5 as a statistically based alternative of contact finding and both models have been compared in terms of correctness at determination of contact numbers and generation of SIR based epidemics.

It has been shown that simulations cases independent of spatial structure deliver similar output when computed by a lattice gas cellular automaton based model or the random contact model, proposing deviations of under 4% at all cases. However, to compare both contact finding models at equal settings, the contact rate parameter k of the RCM has to be set properly in relationship to the collision probability k_{ca} of the LGCA-based model. If the above stated is done correctly, agent contacts are generated at lower computing complexity in the RCM and the module may be used as a part of the epidemic simulation.

However, in some cases, a spatial morphology is needed to compute space dependent scenarios where the position and movement of agents play a crucial role in the epidemic model. In this cases, the CA model has been proven to be the preferable choice for contact generation in mathematical epidemic models.

To reduce the computing time of the LGCA model, an parallel implementation of the LGCA realized with the Computer Unified Device Architecture (CUDA) technology has been presented in chapter 4 of the diploma thesis. The presented algorithm may be used to speed up simulations based on the LGCA method when executed on consumer graphical cards and controlled from the computing environment MATLAB.

It has been demonstrated that an convenient use of GPU resources results in a up to 100 times acceleration of the algorithm in comparison to the CPU execution time, especially for hexagonal grids with high dimension size. Considering the fast development of graphical cards

and constantly increasing amount of streaming processors per GPU chip this difference in time complexity between simulations on the GPU and CPU is expected to increase further.

As this implementation represents a primary approach to porting of the LGCA on the GPU there is potential to improve the presented algorithm and its performance. For example the difference in execution time of single memory transfer and repeated memory transfer executions could be minimized if the second type wouldn't transfer the total amount of grid data after every execution but only a list of contacted agents. Additionally, this list could be created on the device during the execution. Other improvements include further optimization of memory usage and used instructions as well as adjustment of the source code to new generations of CUDA supported GPU devices. Another interesting step would be an implementation of the algorithm in the GPU language *OpenCL* which is an open and non-proprietary language that is able to compile source code compatible to any type of GPU, regardless the manufacturer. Nevertheless, the GPU implementation demonstrates a necessary step towards parallelization of computational tasks in science that is achievable with consumer graphic cards at very low hardware costs. It can be expected that the use of general purpose GPU computing will spread in the future and parallel architectures will extend closer towards the central processing unit. By that, computing tasks of high complexity may be executed on platforms that are best suitable for the desired solution.

In general, the results obtained in chapter 3 of this diploma thesis deliver useful insights in the generation of agent contacts in epidemic simulations and demonstrate how simulation methods of different type may lead to similar results when key parameters are identified and adjusted to each other. Furthermore, in chapter 5 it has been demonstrated how convenient use of parallel architectures that are available at almost every personal computer may speed up algorithms by more than 100 times. This enables a faster execution of future epidemic simulations and more efficient development of new models.

Bibliography

[1] *E. Bender*, "An introduction to mathematical modeling", Dover books on mathematics, New York, Willey Verlag, pages 13-36, 2000.

[2] *K. Velten*, "Mathematical Modeling and Simulation: Introduction for Scientists and Engineers", Wiley-VCH Verlag GmbH and Co, 2008

[3] *F. Breitenecker* "Introduction to Modeling and Simulation" Course Material, UT Vienna, 2010

[4] *C.M. Macal*, "Model Verification and Validation", Workshop on Threat Anticipation: Social Science Methods and Models, 2005

[5] *K.Miksits, H.Hahn*, "Basiswissen Medizinische Mikrobiologie und Infektiologie", Springer Verlag 2009, ISBN 3540644830

[6] *World Health Organisation*, "The World Health Report 2004",

[7] *J.K. Taubenberger, D.M. Morens*, "1918 Influenza: the Mother of All Pandemics", in: Emergent Infectious Diseases 12(1): 15-22, 2006.

[8] *E. Bonabeau*, "Agent-based modeling: Methods and techniques for simulating human systems", in: Colloquium: Adaptive Agents, Intelligence, and Emergent Human Organization: Capturing Complexity through Agent-Based Modeling, vol. 99, suppl. 3, 2002.

[9] *S. Wolfram*, "A New Kind of Science", in Wolfram Media Inc, 2002.

[10] *D.A. Gladrow*, 'Lattice-gas cellular automata and lattice-Boltzmann models", Springer Verlag, Berlin, Heidelberg, pages 151-162, 2000.

[11] *C.M. Macal, M.J. North*, "Tutorial on agent-based modelling and simulation", in: Journal of Simulation, pages 151-162, 2010.

[12] *C.M. Macal, M.J. North*, "Tutorial on agent-based modelling and simulation - Part 2", in: Proceedings of the 2008 simulation seminar, 2006.

[13] *N. Popper, M. Gyimesi, G. Zauner, F. Breitenecker*, in: Proceedings of the 2006 Winter Simulation Conference, 2008.

[14] *G.Schneckenreither, N.Popper, G.Zauner, F. Breitenecker*, "Modelling SIR-Type Epidemics by ODEs, PDEs, Difference Equations and Cellular Automata - A Comparative Study", Simulation Modelling Practice and Theory 2008, Volume 16, Issue 8, 1014-1023

[15] *S. Yakowitz, J. Gani, R. Hayes*, "Cellular Automaton Modeling of Epidemics", Applied Mathematics and Computation, Volume 40, Issue 1, 1990, Pages 41-54

[16] *H. Fukś, A. Lawniczaki*, "Individual-based Lattice Model for Spatial Spread of Epidemics", Discrete Dynamics in Nature and Society 2001, Volume 6, Issue 3, 191-200

[17] *S. Emrich*, 'Comparison of Mathematical Models and Development of a Hybrid Approach for the Simulation and Forecast of Influenza Epidemics within Heterogeneous Populations'", in Diploma thesis, UT Vienna, 2007.

[18] *F. Hayot, M. Mandal, P. Sadayappan*, "Implementation and Performance of a Binary Lattice Gas Algorithm on Parallel Processor Systems", Journal of Computational Physics 80, 277-287 (1989)

[19] *P. Sarkar*, "A Brief History of Cellular Automatas", ACM Computing Surveys, Vol. 32, No. 1, (2000)

[20] *J. Yepez, G.P. Seeley, N.H. Margolus*, "Lattice-Gas Automata Fluids on Parallel Supercomputers", Technical Report 11/1993

[21] *L. Žaloudek, L. Sekanina, V. Šimek*, "GPU Accelerators for Evolvable Cellular Automata", Computation World 2009.49, 533-537

[22] *S. Gobron, H. Bonafos, D. Mestre*, "GPU Accelerated Computation and Visualization of Hexagonal Cellular Automata", ACRI '08 Proceedings of the 8th international conference on Cellular Automata for Reseach and Industry, 512-521

[23] *M.G.B. Johnson, D.P. Playne, K.A. Hawick*, "Data-Parallelism and GPUs for Lattice Gas Fluid Simulations", Massey University, Technical Report CSTN-109

[24] *T.i Kobori, T. Maruyama and T. Hoshino*, "High Speed Computation of Lattice Gas Automata with FPGA", Lecture Notes in Computer Science, 2000, Volume 1896/2000, 801-804

[25] *J. Sanders, E. Kandrot*, "CUDA by Example - An Introduction to General-Purpose GPU Programming", Addison-Wesley Professional, 2010, ISBN 978-0131387683

[26] *D. Kirk, W. Hwu*, "Programming Massively Parallel Processors", Morgen Kaufman 2010, ISBN 978-0-12-381472-2

[27] *MathWorks ltd*, "Parallel Computing Toolbox Documentation", MATLAB 2011a Documentation

[28] *Nvidia Corporation*, "Compute Unified Device Architecture Programming Guide Version 3.2", http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[29] *Nvidia Corporation*, "CUDA C Best Practices Guide Version 3.2", http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

[30] *Nvidia Corporation*, "Accelerating MATLAB with CUDA", Available at the official CUDA website

[31] *AMD*, "The AMD Fusion Family of APUs", fusion.amd.com

[32] *H. Wong, M. Papadopoulou*, "Demystifying GPU Microarchitecture through Microbenchmarking", Analysis of Systems and Software 2010, 235 - 246

[33] *J. Borkowski, M. Rulak*, "Parallel Simulation of Liquid Flow Using LGA Model", International Conference on Parallel Computing in Electrical Engineering 2002, pp.273

Source Code of Parallel Implementation

Listing 1: Host Code: cellular_gpu.m

```
classdef cellular_gpu < handle
    properties
        ca_dichte; $density of the grid
        ca_groesse; $size of the grid
        ca_updates; $number of updates
        ca_turn;
        modus; %1=calculate size from density, 2=calculate density from size
    end
    methods
        function obj = cellular(obj,modus,ca_dichte,ca_groesse,ca_updates) %initializing
            obj.ca_dichte=ca_dichte;
            obj.ca_groesse=ca_groesse;
            obj.modus=modus;

            obj.ca_updates=ca_updates;
        end

        function kontakte=genKontakte(obj,id_teilnehmer)
            kontakte{2}=[];
            kontakte{3}=[];
            kontakte{4}=[];
            kontakte{5}=[];
            kontakte{6}=[];

            partikel_anz = double(length(id_teilnehmer));
            if(obj.modus==1)
                ca_size = int32(ceil(sqrt(partikel_anz/(6*obj.ca_dichte))));
            elseif(obj.modus==2)
                ca_size = obj.ca_groesse;
            end

            reset(parallel.gpu.GPUDevice.current());
            kernel=parallel.gpu.CUDAKernel('lgca_agents_shared.ptx','lgca_agents_shared.cu','runLGCA');
```

```
34              kernel.ThreadBlockSize = [20 20];
35              kernel.GridSize = [round(double(ca_size/20)) round(double(ca_size/20))];
36
37              ca_matrix = zeros(ca_size,ca_size,6,'uint32');
38              ca_matrix(1:partikel_anz) = id_teilnehmer(:);
39              ca_matrix(1:ca_size*ca_size*6) = ca_matrix(randperm(ca_size*ca_size*6));
40
41              Db1 = parallel.gpu.GPUArray( uint32(ca_matrix(:,:,1)) );
42              Db2 = parallel.gpu.GPUArray( uint32(ca_matrix(:,:,2)) );
43              Db3 = parallel.gpu.GPUArray( uint32(ca_matrix(:,:,3)) );
44              Db4 = parallel.gpu.GPUArray( uint32(ca_matrix(:,:,4)) );
45              Db5 = parallel.gpu.GPUArray( uint32(ca_matrix(:,:,5)) );
46              Db6 = parallel.gpu.GPUArray( uint32(ca_matrix(:,:,6)) );
47
48              for zeit_ca = int16(1):obj.ca_updates
49                  ind=findn(ca_matrix);
50                  sind=sortrows(ind,[1 2]);
51
52                  x=sind(1,1);
53                  y=sind(1,2);
54                  j=0;
55                  ids=zeros(1,6);
56                  %ids=[];
57                  for i=1:size(sind,1)
58                      if(sind(i,1)==x && sind(i,2)==y)
59                          j=j+1;
60                          ids(j)=ca_matrix(sind(i,1),sind(i,2),sind(i,3));
61                      else
62                          if(j==2)
63                              kontakte{2} = [ kontakte{2} ; ids(1,1:2) ];
64                          elseif(j==3)
65                              kontakte{3} = [ kontakte{3} ; ids(1,1:3) ];
66                          elseif(j==4)
67                              kontakte{4} = [ kontakte{4} ; ids(1,1:4) ];
68                          elseif(j==5)
69                              kontakte{5} = [ kontakte{5} ; ids(1,1:5) ];
70                          elseif(j==6)
71                              kontakte{6} = [ kontakte{6} ; ids(1,1:6) ];
72                          end
73                          ids=zeros(1,6);
74                          ids(1)=ca_matrix(sind(i,1),sind(i,2),sind(i,3));
75                          x=sind(i,1);
76                          y=sind(i,2);
77                          j=1;
78                      end
79                  end
80
81                  [Db1, Db2, Db3, Db4, Db5, Db6]=feval( kernel, Db1, Db2, Db3, Db4, Db5,...
82          Db6, int16(ca_size), int16(ca_size) );
```

```
83
84                      ca_matrix(:,:,1)=gather(Db1);
85                      ca_matrix(:,:,2)=gather(Db2);
86                      ca_matrix(:,:,3)=gather(Db3);
87                      ca_matrix(:,:,4)=gather(Db4);
88                      ca_matrix(:,:,5)=gather(Db5);
89                      ca_matrix(:,:,6)=gather(Db6);
90
91                  end
92              end
93          end
94      end
```

Listing 2: Device Code: lgca_agents_shared.cu

```
1   #ifndef _LGCA_KERNEL_H_
2   #define _LGCA_KERNEL_H_
3   #include <stdio.h>
4
5   // Shared Memory Size - Must be identifical with GridDim
6   #define SHARED_DIM_X 22
7   #define SHARED_DIM_Y 22
8
9   __shared__ unsigned int as[(SHARED_DIM_X)*(SHARED_DIM_Y)];
10  __shared__ unsigned int bs[(SHARED_DIM_X)*(SHARED_DIM_Y)];
11  __shared__ unsigned int cs[(SHARED_DIM_X)*(SHARED_DIM_Y)];
12  __shared__ unsigned int ds[(SHARED_DIM_X)*(SHARED_DIM_Y)];
13  __shared__ unsigned int es[(SHARED_DIM_X)*(SHARED_DIM_Y)];
14  __shared__ unsigned int fs[(SHARED_DIM_X)*(SHARED_DIM_Y)];
15
16  __device__ unsigned int getAdr( int r, int c, int m, int n) {
17      r = ( r >= m ? r - m : r );
18      r = ( r < 0 ? r + m : r );
19
20      c = ( c >= n ? c - n : c );
21      c = ( c < 0 ? c + n : c );
22      return c * m + r;
23  }
24
25  __global__ void runLGCA(unsigned int* b1, unsigned int* b2, unsigned int* b3,
26          unsigned int* b4, unsigned int* b5, unsigned int* b6, int m, int n)
27  {
28      int x = blockIdx.x * blockDim.x + threadIdx.x;
29      int y = blockIdx.y * blockDim.y + threadIdx.y;
30      int tidx=threadIdx.x+1;
31      int tidy=threadIdx.y+1;
32
33      // Adressing in shared memory
34      int linidx = getAdr(tidx,tidy, SHARED_DIM_X, SHARED_DIM_Y);
```

```
35      // Adressing in global memory
36      int linidxG = getAdr(x,y,m,n);
37
38      // Actual cell
39      as[linidx] = b1[linidxG];
40      bs[linidx] = b2[linidxG];
41      cs[linidx] = b3[linidxG];
42      ds[linidx] = b4[linidxG];
43      es[linidx] = b5[linidxG];
44      fs[linidx] = b6[linidxG];
45
46      // Border cells
47      if ( threadIdx.x == 0 && threadIdx.y != blockDim.y − 1 )
48      {
49          fs[getAdr(0, tidy, SHARED_DIM_X, SHARED_DIM_Y)] = b6[getAdr(x−1, y, m, n)];
50          es[getAdr(0, tidy+1, SHARED_DIM_X, SHARED_DIM_Y)] = b5[getAdr(x−1, y+1, m, n)];
51      }
52      if ( threadIdx.x == blockDim.x − 1 && threadIdx.y != 0 )
53      {
54          bs[getAdr(blockDim.x + 1, tidy−1, SHARED_DIM_X, SHARED_DIM_Y)]=b2[getAdr(x+1, y−1, m, n)];
55          cs[getAdr(blockDim.x + 1, tidy, SHARED_DIM_X, SHARED_DIM_Y)]=b3[getAdr(x+1, y, m, n)];
56      }
57      if ( threadIdx.y == 0 && threadIdx.x != blockDim.x − 1)
58      {
59          as[getAdr(tidx, 0, SHARED_DIM_X, SHARED_DIM_Y)] = b1[getAdr(x, y−1, m, n)];
60          bs[getAdr(tidx+1, 0, SHARED_DIM_X, SHARED_DIM_Y)] = b2[getAdr(x+1, y−1, m, n)];
61      }
62      if ( threadIdx.y == blockDim.y − 1 && threadIdx.x != 0)
63      {
64          ds[getAdr(tidx, blockDim.y + 1, SHARED_DIM_X, SHARED_DIM_Y)] = b4[getAdr(x, y+1, m, n)];
65          es[getAdr(tidx−1, blockDim.y + 1, SHARED_DIM_X, SHARED_DIM_Y)] = b5[getAdr(x−1,y+1, m, n)];
66      }
67      if ( threadIdx.x == 0 && threadIdx.y == blockDim.y − 1)
68      {
69          fs[getAdr(0, blockDim.y , SHARED_DIM_X, SHARED_DIM_Y)] = b6[getAdr(x−1, y, m, n)];
70          es[getAdr(0, blockDim.y + 1, SHARED_DIM_X, SHARED_DIM_Y)] = b5[getAdr(x−1 ,y+1, m, n)];
71          ds[getAdr(1, blockDim.y + 1, SHARED_DIM_X, SHARED_DIM_Y)] = b4[getAdr(x , y+1, m, n)];
72      }
73      if ( threadIdx.x == blockDim.x − 1 && threadIdx.y == 0)
74      {
75          as[getAdr(blockDim.x , 0, SHARED_DIM_X, SHARED_DIM_Y)] = b1[getAdr(x, y−1, m, n)];
76          bs[getAdr(blockDim.x +1, 0, SHARED_DIM_X, SHARED_DIM_Y)] = b2[getAdr(x+1, y−1, m, n)];
77          cs[getAdr(blockDim.x +1, 1, SHARED_DIM_X, SHARED_DIM_Y)] = b3[getAdr(x+1, y, m, n)];
78      }
79
80
81      bool a=(as[linidx] ? true : false);
82      bool b=(bs[linidx] ? true : false);
83      bool c=(cs[linidx] ? true : false);
```

```
84      bool d=(ds[linidx] ? true : false);
85      bool e=(es[linidx] ? true : false);
86      bool f=(fs[linidx] ? true : false);
87
88      // Two body collision
89      bool twoCol = ((a&d&~(b|c|e|f)) | (b&e&~(a|c|d|f)) | (c&f&~(a|b|d|e)));
90
91      // Three body collision
92      bool threeCol = (a^b) & (b^c) & (c^d) & (d^e) & (e^f);
93
94      if (twoCol|threeCol)
95      {
96          //Rotate
97          bool direction = (x+y)%2;
98          unsigned int temp;
99          if(direction)
100         {
101             temp=fs[linidx];
102             fs[linidx]=es[linidx];
103             es[linidx]=ds[linidx];
104             ds[linidx]=cs[linidx];
105             cs[linidx]=bs[linidx];
106             bs[linidx]=as[linidx];
107             as[linidx]=temp;
108         }
109         else
110         {
111             temp=as[linidx];
112             as[linidx]=bs[linidx];
113             bs[linidx]=cs[linidx];
114             cs[linidx]=ds[linidx];
115             ds[linidx]=es[linidx];
116             es[linidx]=fs[linidx];
117             fs[linidx]=temp;
118         }
119     }
120
121     __syncthreads();
122
123     // Wait for all collisions to be computed and prepare for writing propagation
124     unsigned int nextA = as[ getAdr(tidx, tidy−1, SHARED_DIM_X, SHARED_DIM_Y) ];
125     unsigned int nextB = bs[ getAdr(tidx+1, tidy−1, SHARED_DIM_X, SHARED_DIM_Y) ];
126     unsigned int nextC = cs[ getAdr(tidx+1, tidy, SHARED_DIM_X, SHARED_DIM_Y) ];
127     unsigned int nextD = ds[ getAdr(tidx, tidy+1, SHARED_DIM_X, SHARED_DIM_Y) ];
128     unsigned int nextE = es[ getAdr(tidx−1, tidy+1, SHARED_DIM_X, SHARED_DIM_Y) ];
129     unsigned int nextF = fs[ getAdr(tidx−1, tidy, SHARED_DIM_X, SHARED_DIM_Y) ];
130
131     // Collission protection
132     __syncthreads();
```

```
133
134     as[linidx] = nextA;
135     bs[linidx] = nextB;
136     cs[linidx] = nextC;
137     ds[linidx] = nextD;
138     es[linidx] = nextE;
139     fs[linidx] = nextF;
140
141     __syncthreads();
142
143     if ( x < n && y < m )
144     {
145         b1[linidxG] = as[linidx];
146         b2[linidxG] = bs[linidx];
147         b3[linidxG] = cs[linidx];
148         b4[linidxG] = ds[linidx];
149         b5[linidxG] = es[linidx];
150         b6[linidxG] = fs[linidx];
151     }
152 }
153
154 #endif // #ifndef _LGCA_KERNEL_H_
```