Parallel Computing. Advances and Current Issues (Parco 2001)
G. Joubert, A. Murli, F. Peters, and M. Vaneschi (Eds.), Imperial College Press, 2002

DISTRIBUTED SIMULATION ENVIRONMENT FOR COUPLED CELLULAR AUTOMATA IN JAVA

MARCUS BRIESEN*AND JÖRG R. WEIMAR†
Institute of Scientific Computing, Technical University Braunschweig, D-38092 Braunschweig, Germany ; E-mail: J.Weimar@tu-bs.de

We describe a software environment for the coupling of different cellular automata. The system can couple CA for parallelization, can couple CA with different lattice sizes, different spatial or temporal resolutions, and even different state sets. It can also couple CA to other (possibly non-CA) simulations. The complete system, which is integrated with the CA simulation system JCASim, is written in Java and and uses remote method invocation as the communication method. The parallel efficiency is found to be satisfactory for large enough simulations.

1 Introduction

The concept of cellular automata is a framework for simulating highly parallel processes. They are very easy to parallelize and are often called "embarrassingly parallel". Usually, a cellular automaton consists of a regular lattice of cells, each of which contains a state from a finite set of states. From one time step to the next, all cells change their state in parallel depending on their current state and the state of a finite set of neighbors. Since the lattice is uniform, cellular automata are well suited only for the simulation of spatial-temporal phenomena in fairly uniform spatial domains. Some non-uniformity can be introduced by the initial conditions, but in other cases it would be better to combine different cellular automata for different regions in space.

In this paper we demonstrate several types of combinations of cellular automata in the context of a cellular automata simulation system written in Java, JCASim[1,2]. In this system, cellular automata can be described in a compact way using Java, or in a specialized CA language, CDL (which is then translated into Java). The system allows the simulation of cellular automata in one, two, and three dimensions, has several built-in boundary conditions, and allows the display of CA using text, colors, or icons.

---

*Present address: disy Informationssysteme GmbH, Stephanienstr. 30, D-76133 Karlsruhe, Germany.

†To whom correspondence should be addressed.

We consider five different types of coupling between cellular automata (CA), which can also be combined:

1.1 Coupling identical CA

If one or more identical CA are coupled through interactions at their borders, this can be used as a parallelization strategy if the simulation of the CA runs on different processors or computers. This option can also be used to model simulation domains which can not efficiently be covered by one rectangle, e.g., an L-shaped domain. Of course, the coupling interface must be able to connect the corresponding cells on the borders of the different sub-regions.

1.2 Coupling CA with different state sets

In order to couple two CA with different state sets, the programmer of the CA must specify how the state of a cell in one CA can be converted into the state of the other CA. In some cases this can lead to a loss of information, but this should be under the control of the designer of the CA. All the facilities required for case 1 are also required here.

1.3 Coupling CA with different spatial or temporal resolution

In some simulations, certain regions need to be resolved much better than others. This can be achieved by using a fine grid in some regions and a coarse grid elsewhere. In order to combine CA with different resolutions, one must provide interpolation routines and ensure the correct synchronization of the different CA. The interpolation is a more difficult problem for cellular automata than for finite difference or finite element methods, since CA are discrete in space and *state*, therefore averaging different cells is not directly possible.

1.4 Composing CA

The preceding cases all describe the coupling of different CA at a common border. A different case is the coupling over the whole space, which can also be viewed as the composition of different CA. In order for this kind of composition to work, the programmer must again provide methods for the translation of one state into the other.

1.5 Coupling CA with different (possibly non - CA) simulations

CA can also be coupled to other simulations that need not be cellular automata. An example could be a solver for an ordinary differential equation, which provides the boundary conditions for a CA.

2 Implementation

The JCASim system is completely implemented in Java. Therefore the coupling extension[3] is also implemented in Java. In order to enable the parallelization in a distributed environment, each CA can run on a separate virtual machine (VM), and communication between them is handled via remote method invocation (RMI).

An alternative would have been to use CORBA. Here we don't need the language independence of CORBA and prefer the RMI approach, as it is in integrated into the language standard. Furthermore, later migration is expected to be possible with automated tools. Another possibility would be to restrict parallelism to shared memory machines, and simply use multithreading. This approach will be added later for improved performance.

2.1 Structure of JCASim

The JCASim system is designed as a consequently object-oriented system. A `CellularAutomaton` has a `Lattice` of a certain dimension and type. The `Lattice` contains `Cells`, and each `Cell` contains the `State`. This `State` is the class written by the user and contains all the information specific to a special CA, apart from configuration options specified in the simulation system. The user writes a subclass of `State` which contains variables to be held in each cell and contains the method for changing the state in one time step. Access to the neighbors is through a method of the `Cell`, `getNeighbors()`.

If a cell accesses a neighbor outside the simulation region, a special object of type `BoundaryHandler` is called to handle this request. There are three basic boundary handlers in JCASim: `PeriodicBoundaryHandler`, `ReflectiveBoundaryHandler`, and `ConstantBoundaryHandler`. They return the appropriate state of the lattice at the periodic or reflected position or a special constant state.

For coupled CA, a different `BoundaryHandler` is introduced, which is able to get a cell from the remote CA. This `RemoteBoundaryHandler` cooperates with a `BoundaryServer` on the remote site. The `BoundaryHandler` and the `BoundaryServer` also need to ensure the proper synchronization of the CA. Each CA can only start the next iteration if the cell data are not required by

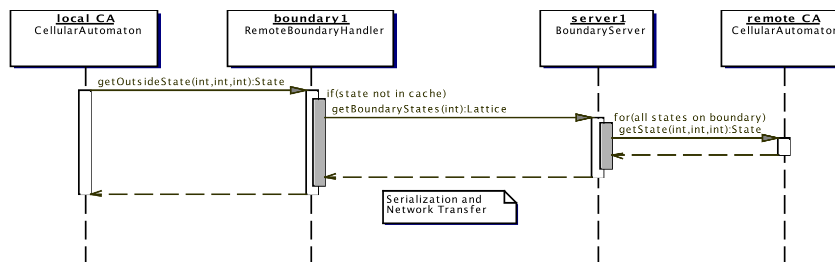

Figure 1. Sequence diagram for the transfer of cells from one CA to another.

another coupled CA. The synchronicity requirement is somewhat relaxed by the fact that each CA keeps a copy of the old state for access to the neighbors during the state transition.

2.2 Remote Data Transfer

The transfer of data from one CA to the other goes through a number of stagesas shown in Figure 1. Within one time step, an access to an outside cell of CA 1 leads to a call to **`getOutsideState`** of the appropriate **`RemoteBoundaryHandler`**. The **`RemoteBoundaryHandler`** (RBH) checks whether the requested cell state is available in the cache. If so, it is returned from the cache (and no remote communication is necessary). If not, a remote method invocation to the corresponding **`BoundaryServer`** on the remote machine is initiated. The RBH requests not just one cell, but a whole slice in the assumption that most other boundary cells will also be requested within the same time step. The request is serialized (converted into a byte stream) and sent over the network to the remote **`BoundaryServer`**, where the request is unpacked. The **`BoundaryServer`** then creates a temporary object of class **`SimpleLattice`** which contains just those cells of the lattice requested by the RBH. This object is again serialized and sent back to the requesting RBH, which then unpacks the results and stores it in its cache and can finally deliver the requested cell state.

Most of the time required for the data exchange is spent in the serialization and deserialization of the SimpleLattice transferring cell states. This could be improved by using the **`Externalizable`** interface instead of the built-in **`Serializable`** mechanism. The **`Externalizable`** interface simply replaced the automatic conversion of objects to byte-streams by a programmer-controlled conversion. The disadvantage is that the user must provide code

for this operation, while serialization is automated using the java reflection mechanism.

Even more efficient is the case where all processes run on the same multiprocesor. In this case, a simple shared-memory access is sufficient. The current implementation does not yet detect this situation.

2.3 Split boundaries, interpolation, and state set conversion

The case where two CA meet, but do not share a complete border (see Figure 2, left) is handled by another specialized `BoundaryHandler`, a `SplitBoundaryHandler`, which in turn uses `RemoteBoundaryHandler`s to communicate with the remote CA.

The case of different spatial resolutions is also handled by a specialized `InterpolatingBoundaryHandler`.

In order to implement coupling case 2 (different state sets), the `RemoteBoundaryHandler` compares the classes of the local and the remote cells states. If they are not equal, and the local state implements the `StateConversion` interface, then the remote boundary handler creates new states of the local class and asks them to import the data from the remote cells.

2.4 Synchronization

Two coupled CA must be synchronized so that each request for states from the other CA can return states at the correct time step. If one CA advances too fast, the states requested by the other CA are not available any more (states from later times are stored in their place). On the other hand, the number of requests is not predetermined, since a CA might during some time steps not request any outside neighboring cells at all. Therefore separate synchronization calls are inserted, where each CA announces to its `RemoteBoundaryHandlers` the intention to proceed to the next time step. The RBH can delay this progress until the corresponding `RemoteServer` agrees that the next time step can be executed.

3 Initialization

The different CA, boundary handlers, and boundary servers are initially set up by a central configurator. This configurator has a graphical user interface for setting up the simulation In the configurator it is possible to add a new cellular automaton to the simulation. The boundaries can be connected automatically

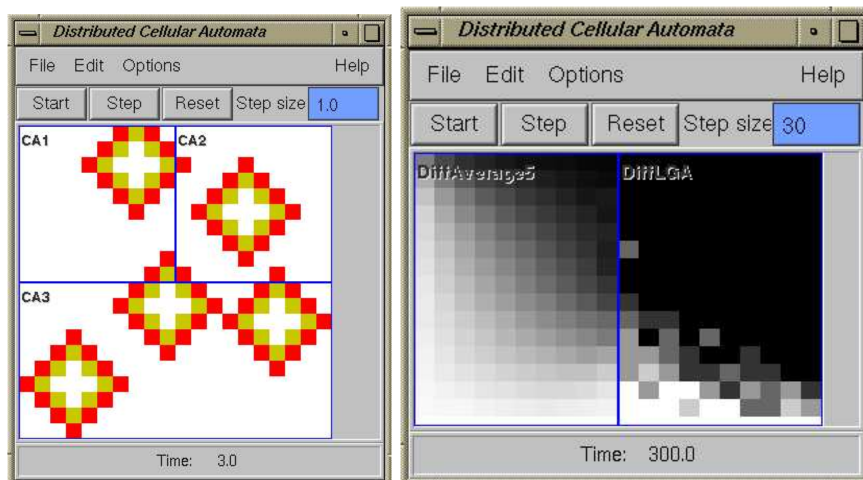

Figure 2. Left: Greenberg-Hastings CA testing general communication and boundary splitting. Right: Diffusion with two different approaches

or explicitly. The configuration of a simulation can be saved in XML-format, which can then also be edited with a text-editor or with a automated tools to create larger simulations.

4 Simulation

The simulation of coupled cellular automata is controlled by a user interface which can display all of the CA together. The user specifies a time until which the simulation should be run. All the coupled CA receive this information and execute the necessary transition steps to reach this given time. The synchronization is strictly local, i.e., neighboring regions control each other. When the predetermined target time is reached, the controller interface is notified. The GUI can then copy the cell states for display to the user.

4.1 Test-cases

As a first test (Figure 2, left), we coupled three CA of the same type to verify the correct exchange at the (split) border. We selected a Greenberg-Hastings CA in which any error in the coupling can be easily spotted.

As an example for coupling different types of CA, we show two differ-

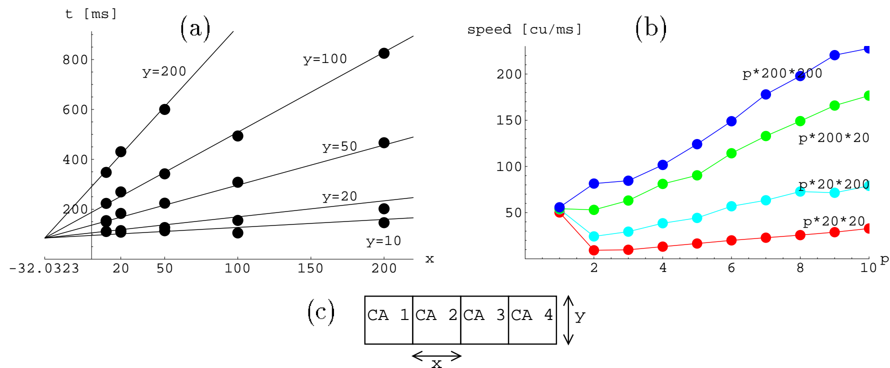

Figure 3. Measurement results: (a) Time in milliseconds per time step to simulate coupled CA with different sizes **x** and **y** (with 4 processors). (b) Speedup for different sizes (where the problem size scales tith the number of processors p). (c) Configuration used.

ent cellular automata simulating diffusion. One automaton uses the lattice gas approach, the other uses a finite difference averaging with probabilistic rounding [4]. Figure 2(right) shows the simulation after 300 time steps.

4.2 Speed

The speed of the simulation is influenced by a number of factors. To obtain a reasonable simulation speed in this Java based system, a number of recommendations should be followed[2]. Most importantly, no object should be created in the inner simulation loop. For CA requiring random numbers, an efficient random number generator should be used. Besides these recommendations, recent Java virtual machine implementations come with just-in-time compilers which are quite efficient.

The coupling interface introduces an additional overhead. This overhead comes from the serialization, which is used to package the border cells, and the remote method invocation, which includes the network communication latency. The current implementation also does not heed the above recommendation to avoid all object creations (Java serialization creates new objects when de-serializing). To measure the overhead, we performed a number of simulations on a Linux-cluster (Pentium 266). The simulations were performed using a typical CA (with diffusion and using random numbers extensively) and different sizes of the CA. We report the simulation times for four processes and indicate the size of the data chunk for each process. In this simulation the

processes are arranged in a linear array and the results are shown in Figure 3. The internal calculation of the transition function takes $0.032ms$ per cell. The border exchange introduces a delay of $85ms$ plus $1.02ms$ per border cell. The constant delay is approximately equal to the variable delay for 82 cells, and the total delay for exchanging 82 cells is approximately equal to the time required to update 82×64 cells. Therefore this approach is only useful for parallelization of large grids.

The speedup (with growing problem size) is nearly linear for two up to 10 processors, since there is no sequential overhead in the synchronization. The border exchange can be improved using the `Externalizable` interface of Java, which reduces the communication time per cell by 25%. The constant overhead could probably be improved by careful tuning of the synchronization messages.

5 Conclusion

We have demonstrated an extension to the cellular automata simulation system JCASim, which allows the coupling of different CA. The CA can have different state sets and different spatial or temporal resolutions. The coupling uses the Java RMI mechanism and is mainly geared towards flexibility as opposed to speed. Parallelization can be performed using this mechanism, but is efficient only for large enough simulations. The JCASim sytem is available at `http://www.jcasim.de/`.

References

1. Uwe Freiwald. Eine Java - Simulationsumgebung für Zellularautomaten. Diplomarbeit, Inst. of Scientific Computing, Techn. University Braunschweig, 1999.
2. Jörg R. Weimar and Uwe Freiwald. JCASim – a Java system for simulating cellular automata. In S. Bandini and T. Worsch, editors, *Theoretical and Practical Issues on Cellular Automata (ACRI 2000)*, pages 47–54, London, 2000. Springer-Verlag.
3. Marcus Briesen. Eine verteilte Simulationsumgebung für gekoppelte Zellularautomaten in Java. Diplomarbeit, Inst. of Scientific Computing, Techn. University Braunschweig, 1999.
4. Jörg R. Weimar. Simulating reaction-diffusion cellular automata with JCASim. In T. Sonar, editor, *Discrete Modelling and Discrete Algorithms in Continuum Mechanics*, page (to be published). Logos-Verlag, Berlin, 2001.