25

Design and Implementation of CAOS: An Implicitly Parallel Language for the High-Performance Simulation of Cellular Automata

Clemens Grelck[1] and Frank Penczek[2]
[1]*University of Amsterdam*
[2]*University of Hertfordshire*
[1]*The Netherlands*
[2]*United Kingdom*

1. Introduction

Cellular automata are a powerful concept for the simulation of complex systems; they have successfully been applied to a wide range of simulation problems Boccara et al. (1993); Canyurt & Hajela (2005); D'Ambrosio et al. (2007); Ermentrout & Edelstein-Keshet (1993); Georgoudas et al. (2007); Guisado et al. (2006); Nagel & Schreckenberg (1992); Popovici & Popovici (2002); Stevens et al. (2007). This work is typically done by scientists who are experts in their field, but generally not experts in programming and computer architecture. Programming complex simulations both correctly and efficiently quickly turns into a painful implementation venture distracting from the far more interesting aspects of the simulation problem itself or the simulated subject matter.

Current advances in computer architecture make the situation even worse. Abundance of parallel processing power through multicore technology requires parallelisation of simulation software in order to effectively use even standard laptop and desktop computing machinery, not to mention clusters of workstations and fully-fledged supercomputing equipment. This situation confronts our dear simulation scientist not only with the task of writing a fairly efficient sequential program, but exposes him or her to the notorious hazards of parallel programming Amarasinghe (2008); Chapman (2007); Gabb et al. (2009). Getting synchronisation and communication requirements correct and deterministic is know to be a far from trivial task, but in fact the problem is even more intricate. Today's hardware reality quickly creates a multi-level granularity problem with different communication and synchronisation abstractions and very different latency/throughput characteristics on each level, from networks interconnecting geographically separated compute centers to multiple cores within the same processor. An experienced programmer can certainly solve all these issues with sufficient time and resources, but the point we make is that scientist we have in mind should not devote his time to this, but rather work on improving his or her model, etc.

The model of cellular automata naturally lends itself to parallel execution following a data parallel approach. This holds not only for multicore processors on the desktop, but likewise for clusters of workstations and parallel computers, in other words on all levels of today's multi-level compute environments. Yet surprisingly little support for programming cellular automata with a focus on high-performace simulation exists. Simulation software is typically

limited in the complexity of cellular automata that can be described: the number of states per cell and the state transition function are typically more oriented towards demonstrations of Conway's game of life Conway (1970) than towards real-life simulations. Furthermore, many approaches seem to focus on the visual aspects of cellular automata rather than simulation performance. Programming complex cellular automata in general-purpose programming languages is not extremely difficult, but it does require substantial programming skills. This hinders the effective utilisation of cellular automata by scientists who are experts in their field, but not necessarily experts in programming.

We propose a new domain-specific programming language named CAOS (Cells, Agents and Observers for Simulation) that is tailor-made for programming simulation software based on the model of cellular automata. Since it is restricted to this single purpose, CAOS provides the scientist with support for the rapid prototyping of complex simulations on a high level of abstraction. Nevertheless, the CAOS compiler fully automatically generates portable and efficiently executable code for a wide range of architectures. We support both shared memory systems through OPENMP Dagum & Menon (1998) and distributed memory systems through MPI Gropp et al. (1994). Both approaches can easily be combined having the compiler generate multithreaded OPENMP code within MPI processes for hybrid architectures. Thus, CAOS not only supports individual multicore processors, but the whole range of computer architecture from laptop processors to supercomputing installations. CAOS allows scientists to harness the potential compute power of both small-scale and large-scale parallel computers for complex simulations with little or even no expertise in parallel programming and computer architecture.

The remainder of this chapter is organised as follows: In Section 2 we introduce the language design of CAOS. Section 4 outlines principles of our implementation while Section 3 provides a brief explanation of the CAOS tool chain. Section 5 discusses a number of runtime performance related experiments. We address related work in Section 6 and conclude in Section 7.

2. CAOS language design

A CAOS program implements all aspects of a cellular automaton simulation. It defines the layout of a multi-dimensional grid of cells, its initialisation (which may also be read from a file) and the behaviour of the cells in the form of a potentially non-trivial state transition function. It also defines how and when snapshots of the simulation are taken and saved. Each of these aspects is implemented in a dedicated CAOS program section. Thus, a CAOS program is organised into a sequence of sections as shown in Fig. 1.

$$\textit{CAOS Program} \quad \Rightarrow \quad \textit{Declarations Grid Cell Init Behaviour} \left[\textit{Observer}\right]^*$$

Fig. 1. General structure of CAOS source files

The *declaration* section contains a number of global decalarations referring to user-defined types, compile time constants and runtime program parameters. Next comes the *grid* section with the definition of the grid, which may have any number of axes, sizes and different boundary conditions. The *cell* section defines the attributes making up each cell's state. The *initialisation* section defines initial values for cells and boundaries. Most importantly, the *behaviour* section defines the state transition function including the definition of neighbourhoods in the cellular automaton. And, last not least, the *observer* section defines how and when snapshots of the cellular automaton are saved to disk while the simulation is running and/or after it has been completed, depending on the concrete application requirements. In the sequel, we will look into each of these sections in greater detail.

2.1 Global declarations

Unlike the other sections the global declaration section itself is a sequence of subsections declaring different objects to be used throughout the remainder of the program. Fig. 2 gives an overview and defines the exact syntax.

Declarations	⇒	⌈*Enum* ⌉* ⌈*Constant* ⌉* ⌈*Param* ⌉* ⌈*Extern* ⌉*
Constant	⇒	**const** *Type Id* = *value* **;**
Param	⇒	**param** ⌈*"String"* ⌉ *Type Id* = *value* **;**
Enum	⇒	**enum** *Id* { *Id* ⌈, *Id* ⌉* } **;**
Extern	⇒	**extern** *Type Id* (⌈*Type* ⌈,*Type* ⌉* ⌉) **;**
Type	⇒	**bool** | **int** | **double**

Fig. 2. CAOS global declaration syntax

The first set of declarations refers to types. CAOS is not a particularly versatile language when it comes to types and type constructors. We deliberately restrict ourselves here; extending the language design by additional machine-supported basic types and standard type constructors like records is merely a matter of engineering, not of language design. In essence, CAOS supports three built-in types: Boolean values (`bool`), machine word integer numbers (`int`) and double precision floating point numbers (`double`). The only type constructor CAOS supports for now are symbolic enumeration types as known from C. We do this to advocate a symbolic style of programming in contrast to a machine-oriented one. Taking Conway's famous Game of Life as a running example, we would suggest to define an enumeration type `dead_or_alive` rather than encoding these different states in Boolean or integer values
The `const` keyword marks the second kind of global declarations; it is used to declare an identifier with a compile time constant value. Again we recommend to define symbolic constants for relevant numerical values to improve the readability of code.
In addition to (compile time) constants CAOS also features simulation *parameters*. These are compile time symbolic values, but runtime constants initialised during program startup. Parameters are declared using the `param` keyword. Like a constant a parameter has a type, a name and a (default) value. The difference is that the given value indeed only is a default value. This default value may be overwritten upon startup of a simulation through a command line parameter. Parameters are a very useful feature if a simulation needs to be run several times for different parameter sets, e.g. from a shell script. Typical applications for parameters are the initialization of

- values of cell components,
- used variables in the cell behavior,
- time steps for observations,

but many others are possible. The keyword `param` may be followed by an optional string enclosed in quotation marks to compile a short description of the parameter into the executable. The information given here will be displayed within the help text that a compiled CAOS program displays if `--help` is passed as command line parameter.
It is possible to use external functions in a CAOS program via the `extern` keyword and a C-style function prototype. The main intended use of this feature is to make mathematical

libraries available to CAOS program to be used in the definition of the state transition function. Fig. 4 shows a some example declarations in a CAOS implementation of Conway's famous Game of Life.

2.2 Grid layout definition

The main characteristic of cellular automaton based simulation is the existance of a multidimensional grid of cells. The grid of cells in CAOS may indeed comprise an arbitrary number of dimensions, each of a potentially different size. The keyword `grid` marks the beginning of this section; Fig. 3 shows the complete syntax.

Grid	⇒	**grid** *Axis* [**,** *Axis*]* **;**
Axis	⇒	*Id* **:** *Size* **:** *Id* **<.>** *Id* **:** *Boundary*
Size	⇒	*IntConstant* | *Id*
Boundary	⇒	**static** | **cyclic**

Fig. 3. CAOS grid layout definition syntax

A grid topology specification is a comma-separated list of axis specifications. Each axis specification consists of four parts separated by colons. Firstly, we have an identifier that names the axis. The second part defines the size of the grid along the given axis. This can either directly be defined by an integer constant or indirectly through a symbolic identifier, thus making use of the constant or parameter mechanism explained before. In particular, the grid size can but need not be fixed at compiler time. Through the parameter mechanism, this vital simulation parameter can easily be supplied at runtime.
The following two identifiers separated by the special symbol `<.>` define names for accessing neighbouring grid cells in direction of decreasing and of increasing indices, respectively. There use will be discussed in detail in Subsection 2.5. The last part defines the boudary condition of the grid, which can either be `static` or `cyclic`. With cyclic boundary condition, the cells on one boundary do have the cells on the other boundary of that axis as their neighbours. With static boundary condition additional constant boundary cells are added to the grid to ensure that all proper cells have a complete neihbourhood. The CAOS example in Fig. 4 continues with a 2-dimensional grid layout declaration where the boundary conditions are cycly, the extent along the x -axis is fixed to 256 cells and the extent along the y-axis defaults to 256 cells, but can be overwritten by a command line parameter at program startup. The directions from any one cell to its neighbours in the grid are named *left* and *right* along the x-axis and *up* and *down* along the y-axis.

```
enum dead_or_alive {dead, alive};
const int x-size = 256;
param "Y grid size" int y-size = 256;

grid:X:x-size:left<.>right:cyclic, Y:y-size:up<.>down:cyclic;

cell {
  dead_or_alive state;
}
```

Fig. 4. CAOS example Game of Life implementation (declaration part)

2.3 Cell attribute definition

After defining the size and topology of the grid, it needs to be populated with cells. The `cell` section defines the attributes of each cell, which can be drawn from the set of built-in types `int`, `double` and `bool` plus previously defined enumeration types. As shown in Fig. 5, syntactically the cell section very much resembles record definitions in imperative languages or the attribute sections of class definitions in object-oriented langauges.
Cell atrributes are readable by other cells from the neighborhood. For example, our running example in Fig. 4 has a single attribute that we simply call *state*, which can be either *dead* or *alive* according to the previous definition of the enumeration type (and the definition of the Game of Life). In general, cells can have a number of different attributes of different types supporting complex state spaces.

Cell ⇒ **cell** { ⌈ *Attribute* ⌋+ }

Attribute ⇒ *Type Id* **;**

Fig. 5. CAOS cell attribute definition syntax

2.4 Grid initialisation

Init ⇒ **init** ⌈*Selector*⌋ { ⌈ *Assignment* ⌋+ }

Selector ⇒ [*Id* ⌈^ *Id*⌋*]

Assignment ⇒ *Id* = *Expr* **;**

Fig. 6. CAOS initialisation section syntax

Before the simulation of the first time step the cells on the grid are initialised. This initial state is defined through a `init` section as shown in Fig. 6 or it may be read in from a file as well. All components of a cell, which are defined in the `cell` section, appear as left hand side of an assignment in the `init` section. Of course, it is possible to leave components uninitialised, but this may obviously lead to erroneous and unpredictable behaviour.
The keyword `init` may be followed by a *selector* to initialize boundary cells at static boundaries. Assume that we would change the grid definition of our example in Fig. 4 to

```
grid:X:1..40:left<.>right:static,Y:1..40:up<.>down:static;
```

Then, to assign values to the cells on the lower boundary of the first dimension, `init[left]` would be used. It is also possible to combine direction specifier. Accordingly, `init[right ^ down]` would initialize the cells that belong to the upper boundary of the first *and* the second dimension.

2.5 Simple state transition functions

In CAOS the state transition function of the cellular automaton is defined by he `behaviour` section. Unlike most cellular automata simulation systems, CAOS provides a fully-fledged, structured imperative programming language to define complex state transition functions. As shown in Fig. 7, we base CAOS on the syntactic foundations of C to faciliate familiarisation.
A simple CAOS behaviour section consists of a sequence of assignments preceded by declarations of local variables. Whereas the declaration of local variables syntactically very

Behaviour ⇒ **behaviour** { ⌈*LocalVarDecl*⌉* ⌈*Instruction*⌉* }

LocalVarDecl ⇒ *LocalType Id* **;**

LocalType ⇒ *Type* | **dir**

Instruction ⇒ *Assignment*

Assignment ⇒ *Id* = *Expr* **;**

Expr ⇒ *IntConstant* | *DoubleConstant* | *BoolConstant*
| (*Expr*) | *MonOp Expr* | *Expr BinOp Expr*
| *Id* (⌈*Expr* ⌈ **,** *Expr*⌉* ⌉)
| *Id* ⌈ [*Id* ⌈^ *Id*⌉*] ⌉

Fig. 7. CAOS state transition function syntax

much resembles the declaration of cell attributes in the cell section, the meaning is quite different. Cell attributes form the basis of the cellular automaton, are recomputed in each simulation step and can be read be neighbouring cells. In contrast, local variables in the behaviour section are not more than symbolic placeholders for intermediate values; their meaning is strictly limited to one instantiation of the state transition function.

Like the cell attributes, a local variable can be of any of the built-in types or of any of the previously defined enumeration types. What again separates local variables from cell attributes is the existance of one more built-in type: the direction type `dir`. The values of this type are the direction identifiers introduced in the grid construct. The only operations available on direction values are equality, inequality and the concatenation operator hat. With these restrictions direction values are first-class citizens of CAOS.

Left hand side variables in assignments can be either cell attributes or local variables. In the former case the new value of that attribute is defined; in the latter case the assignment has no effect, unless the local variable occurs in a subsequent expression that actually defines an attribute. Repeated assignment to the same cell attribute or local variable simply overwrites the previous value.

Right hand side expressions are made up of identifiers, the usual unary and binary operator applications and function applications. CAOS supports all built-in operators of C and the same associativities and priorities as in C apply. As CAOS in its current state does not support the definition of functions, function applications refer to external functions declared in the declaration section using the `extern` keyword.

Identifiers in expression position may either refer to cell attributes or to local identifiers. The latter case requires a preceding assignment to the local variable, otherwise its value and, hence, the value of the expression is undefined. If an identifier refers to the cell state, the value of the attribute is always the previous iteraions's value, even if an assignment to the same cell attribute precedes the occurence of the identifier.

A specialty of CAOS is the symbolic definition of neighbourhoods, more precisely the way the previous iteration's state of neighbouring cell's attributes are accessed. While a plain cell attribute identifier refers to the previous value of this attribute in the current cell, neighbouring cells can be accessed through symbolic selectors that make use of the direction identifiers defined together with the grid layout in the grid definition section. To support complex neighbourship relationships, multiple direction specifiers can be combined using the hat operator.

```
grid:X:100:left<.>right:cyclic, Y:100:up<.>down:cyclic;

cell {
  double state;
}

behaviour {
  double sum;
  sum = state + state[up^left] + state[up] + state [up^right]
              + state[left] + state[right]
              + state[down^left] + state[down] + state [down^right];
  state = sum / 9.0;
}
```

Fig. 8. CAOS example comuting average with eight neighbouring cells

Fig.8 illustrates the use of selectors using a simple CAOS program, where the state is a made up of a floating point number, the grid is two-dimensional with cyclic boundary conditions, and the state transition function computes the arithmetic mean between previous value and those of the eight neighbouring cell's values.

2.6 Control flow constructs

Simple state transition functions made up of sequences of assignments are too restricted to define most interesting cases. Even Conway's Game of Life, despite all its simplicity, cannot defined in this restricted setting. For general-purpose applicability CAOS supports a number of control flow manipulating constructs, some of which are directly borrowed from other languages (more precisely from C as far as syntax is concerned), some are tailor-made for CAOS. Fig. 9 defines the additional syntax for CAOS behaviour sections.

Instruction	⇒	*Assignment* | *Cond* | *Switch* | *ForEach*
Block	⇒	*Instruction*
	|	{ ⌈*Instruction*⌉* }
Cond	⇒	**if** (*Expr*) *Block* **else** *Block*
Switch	⇒	**switch** (*Id*) { ⌈*Case*⌉+ ⌈*Default*⌉ }
Case	⇒	**case** *CaseVal* ⌈**,** *CaseVal*⌉* ⌈*Guard*⌉ **:** *Block*
Guard	⇒	| *Expr*
Default	⇒	**default** **:** *Block*
ForEach	⇒	**foreach** (*LocalType Id* **in** *Set* ⌈*Guard*⌉ **)** *Block*
Set	⇒	[*Expr* ⌈**,** *Expr*⌉*]

Fig. 9. CAOS control flow construct syntax

The simplest control flow construct is a conditional or branching construct as supported in one way or another by any programming language. Fig. 10 illustrates its use and shows how Conway's Game of Life can be implemented with just this control flow construct.

```
cell {
  dead_or_alive state;
}

behaviour {
  int counter;

  counter = 0;
  if (state == alive)        counter=counter+1;
  if (state[up] == alive)    counter=counter+1;
  if (state[down] == alive)  counter=counter+1;
  if (state[left] == alive)  counter=counter+1;
  if (state[right] == alive) counter=counter+1;

  if (counter == 2 || counter == 3) {
    state = alive;
  }
  else {
    state = dead;
  }
}
```

Fig. 10. CAOS example implementing the Game of Life behaviour

The second control flow construct is a `switch`-construct. Despite the obvious syntactic similarities, the CAOS `switch` does differ from its C counterpart in essentially two aspects. Each `case`-statement may feature multiple values rather than just one as in C. Accordingly, CAOS does not feature multiple cases sharing the same block of instructions as would be achieved in C by leaving out the `break`-statement in between. Having said that, CAOS does not know the `break`-statement and it will always execute the block of instructions associated with the first case that fits the pattern as defined by the variable following the key word `switch`. The other difference between CAOS and C is that individual cases can be refined by a *guard* expression, which is syntactically is separated from the values by a vertical bar. If the switch variable has one of the values listed by some case, the guard expression is evaluated. This expression must yield a Boolean value. If the value is `true`, the associated block of instructions is executed; otherwise, the execution of the `switch`-construct continues with the next case.
CAOS does not feature any C-like loop constructs, but it does have a related construct, named `foreach`. The `foreach`-construct allows the programmer to define a block of instructions that is executed for each element of a given *set* of elements. Each expression of the set (enclosed by square brackets) is assigned exactly once to the identifier introduced in the construct and the associated block of instructions is executed for this value. As Fig. 9 shows, the set definition may be followed by an optional guard expression that has the same meaning as in the `switch`-construct introduced before. Fig. 11 illustrates the use of `foreach` and `switch` through an alternative implementation of the Game of Life.

2.7 Non-deterministic features

In some scenarios it is desirable to introduce probabilistic behaviour of cells. For this purpose three constructs `forone`, `with` and `choose` are provided as part of the CAOS language. Fig. 12 defines their exact syntax.
The `forone` constructs syntactically very much resembles the (deterministic) `forall`-construct. However, unlike `forall`, `foreach` selects exactly one element from the given set. The element is selected using pseudo-random methods. Once an element from the set is chosen, the (optional) guard expression is evaluated. If it evaluates to `true`,

```
cell {
  dead_or_alive state;
}

behaviour {
  int counter;

  counter = 0;

  foreach (dir d in [left, right, . , up, down] | state[d] == alive) {
    counter = counter + 1;
  }

  switch (counter) {
    case 0,1,4: state = dead;
    case 2,3:   state = alive;
  }
}
```

Fig. 11. CAOS alternative example implementing the Game of Life behaviour using advanced control flow constructs

Instruction ⇒ ...
| *ForeOne* | *With*

ForOne ⇒ **forone** (*Type Id* **in** *Set* ⌈*Guard*⌉) *Block*

With ⇒ **with** (*Expr*) *Block* ⌈ **else** *Block* ⌉

Expr ⇒ ...
| *Choose*

Choose ⇒ **choose** (*LocalType Id* **in** *Set*)

Fig. 12. CAOS syntax of non-deterministic language constructs

the associated block of instructions is executed; otherwise, program execution proceeds to the instruction following the `forone`-construct.
Fig. 13 illustrates the use of the `forone`-construct (and of the other non-deterministic langauge features introduced below). In the first behaviour section one of the four direct neighbours is non-deterministacally chosen and the state of that neighbour defines the new state of the current cell.
If the sole intention of using a `forone` is to non-deterministically choose one value out of a set of values, the `choose`-construct provides a more concise alternative. The `choose`-construct actually is an expression rather than an instruction. It can be used anywhere in expression position; its value is one of the values described by the set, which one is non-deterministc. The second behaviour section in Fig. 13 illustrates the use of `choose`.
Last not least, the `with`-construct introduces the notion of probabilistic execution of code blocks. After specifying a probability $0 \leq p \leq 1, p \in \mathbb{R}$ a block of code will be executed with this probability. With probability $1 - p$ the code block following the key word `else` is executed. The absence of an `else`-block is treated as an empty `else`-block. The third behaviour block in Fig. 13 illustrates the use of `with`. With 70% propability the state of the left neighbour is chosen as new state of the current cell, with 30% propability the state of the right neighbour.

```
behaviour {
  forone (dir d in [left, right, up, down]) {
    state = state[d];
  }
}

behaviour {
  state = choose (int val in [state[left], state[right], state[up], state[down]);
}

behaviour {
  with (0.7) state = state[left];
  else state = state[right];
}
```

Fig. 13. Examples for the use of CAOS non-deterministic features

2.8 Observers

It is paramount for any simulation software to make the result of simulation, and in most cases intermediate states at regular intervals as well, visible for interpretation. Observers serve exactly this purpose. They allow us to observe the values of certain attributes of cells and agents or cumulative data about them (e.g. averages, minima or sums) at certain regular intervals of the simulation or just after completing the entire simulation.
Each observer is connected with a certain file name (not a certain file). The parallel runtime system takes full advantage of parallel I/O both when using MPI and OPENMP as backend. This file system handling is particularly tricky if it is to be hand-coded. An auxiliary tool suite provides a comfortable user-interface to observer data produced through parallel file I/O.

Observer	⇒	*Observeall* | *Observe*
Observeall	⇒	**observeall** (*Filename* **,** *Expr*) { ⌈*ObsAllInstr*⌋+ }
Observe	⇒	**observe** (*Filename* **,** *Expr*) { ⌈*ObsInstr*⌋+ }
ObsAllInstr	⇒	*Type String* = *Expr* **;**
ObsInstr	⇒	*Type String* = *ReduceOp* (*Expr*) **;**
ReduceOp	⇒	**avg** | **min** | **max** | **sum** | **prod** | **all** | **any** | **cnt**

Fig. 14. CAOS observer syntax

There are two conceptual different classes of observers (see Fig. 14 for concrete syntax) that either observe values of cell attributes individually or that apply a reduction operation to all cells and produce scalar results. Both concepts allow the programmer to specify time steps in which the blocks are executed. For this, an Boolean expression depending on the current `timestep` may be specified. It is evaluated in each time step. If the value is true, the associated observation block is executed. Both observer classes require the declaration of a filename. All data gathered by the observer are written to the file specified by that filename. Each entry in an observation block is build up in the same way: First, the type of the result has to be specified, followed by a user-definable identifier of that entry. The identifier is followed by the expression that computes the desired result.

```
observe ("myObserverFile", timestep%10 == 0) {
  int "cellsAlive" = cnt( state == alive);
}

obserevall ("mySnapshot", timestep == 1) {
  int "cellState" = state;
  bool "activeState" = active;
}
```

Fig. 15. Examples for the use of observers

The keyword `observe` denotes the start of an observer that uses reduce operations on selected components. Each line in the observe section yields one value that is written to file. There are eight different reduce operators available:

- `min/max/avg` These operators determine the minimum/maximum/average of the given expression for all cells.
- `sum/prod` The value of the expression for each cell is summed/multiplied up.
- `cnt` If the Boolean expression holds, a counter is increased by one. Eventually, the value of the counter is written to the file.
- `any/all` These operators yield true if the expression is true for all cells/any cell and false otherwise.

Fig. 15 illustrates the use of observers by two simple examples. The first observer observes the number of cells that are in state `alive` after every 10 timesteps.
The keyword `observeall` starts the definition of an observer that saves values for every single cell to the file. The resulting file contains a tuple of results for every cell, representing the results of expressions given in the observer section. Thus, a simple snapshot of the grid is generated by specifying the cell components as results in the observer section in the same order as they appear in the `cell` section. Fig. 15 again provides a simple example.

2.9 Agents

Agents are similar to cells in that they consist of a set of attributes. Agents move from cell to cell; at any step during the simulation an agent is associated with exactly one cell. A cell in turn may be associated with a conceptually unlimited number of agents. Like the cells, agents have a behaviour (or state transition function). The behaviour of an agent is based on its existing state and the state of the cell it resides at as well as all other agents and cells in the neighbourhood as described above. In addition to updating its internal state, an agent (unlike a cell) may decide to move to a neighbouring cell. Conceptually, this is nothing but an update of the special attribute location. Agents also have a life time, i.e. rather than moving to another cell, agents may decide to die and agents may create new agents. Agents are not implemented in the current version of CAOS but are planned for the next release.

3. The CAOS tool-chain

We have implemented a fully fledged CAOS compiler[1] that generates sequential C code. On demand, the grid is automatically partitioned for multiple MPI processes. The process topology including the choice and number of partitioned grid axes are fully user-defined. A default process topology provided at compiler time may be overwritten at program startup. Additionally, each MPI process may be split either statically or dynamically into a

[1] The current version does not yet support agents.

user-defined number of OPENMP threads, provided that the available MPI implementation is thread-safe. Proper and efficient communication between MPI processes including the organisation of halo or ghost cells at partition boundaries is taken care of by the compiler without any user interaction.
The compilation of a CAOS simulation into an executable binary is based on the CAOS compiler *caosC*. Invocation of the compiler is done indirectly via a generic Makefile that implements all required stages from translating CAOS source code to C, choosing an appropriate C compiler and calling the desired compiler with all required options to generate binary code.
The compilation process is parametrised over several options that determine what kind of parallelisation is applied and how many time steps are executed by default. If `make` is called without any options, an extensive help screen is displayed.
The resulting program carries out the simulation steps as shown in Fig. 16. As with

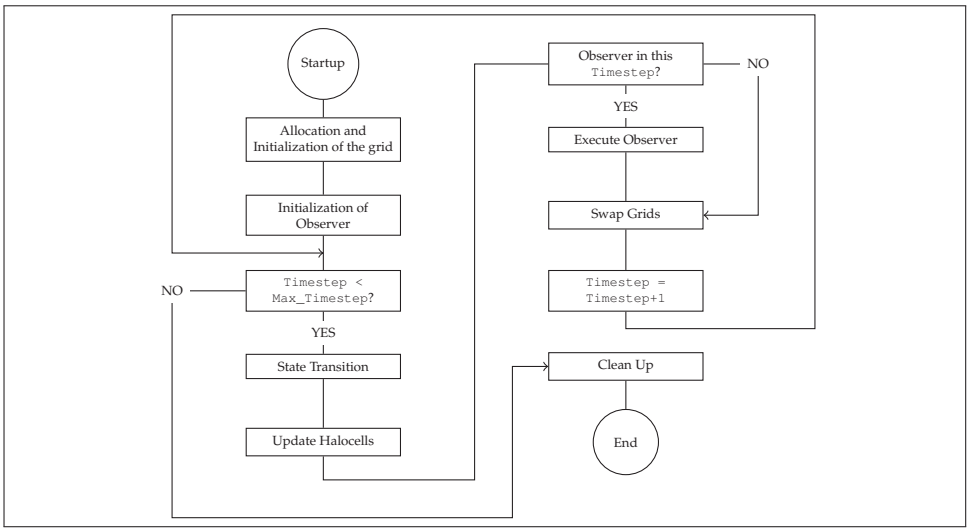

Fig. 16. Program flow chart of a CAOS simulation run

the compilation process, the executable file supports several options too. The parameters influence the runtime behaviour of the program and are as follows:

- `-help` displays a list of all supported parameters
- `-timesteps=`*n* sets the number of simulation time steps to *n*. If this parameter is not given, the compiled-in default number of time steps is executed.
- `-show-progress` shows a visual progress indicator during execution.
- `-infile=`*filename* if this option is present, the initial state of the cell grid is read in from a file *filename*. The `init` block of the CAOS program has no effect in this case.
- `-dimsizes=`*N* this option overrides the compiled-in grid size of the simulation. When using this option, the sizes of all dimensions of the grid have to be specified. Sizes are separated by a lower-case `x`, for example 128 `x` 128.
- `-psizes=`*P* defines the distribution of the grid for each dimension. With this parameter each dimension is divided into as many parts as defined by *P*.Sizes are separated by a

lower-case `x`, for example 2 `x` 2. If a dimension is given a size of 1 no distribution along that dimension is applied. See Fig. 17 for examples.

- `-mpi-psizes=`*P* for simulations that have been compiled into mixed-mode binaries, i.e. simultaneous usage of OpenMP and MPI, this parameter defines the distribution of the grid across MPI processes for each dimension. Each dimension is divided into as many parts as defined by *P*.Sizes are separated by a lower-case `x`, for example 2 `x` 2. If a dimension is given a size of 1 no distribution along that dimension is applied.
- `-omp-psizes=`*P* for simulations that have been compiled into mixed-mode binaries, i.e. simultaneous usage of OpenMP and MPI, this parameter defines the distribution of the grid across OpenMP threads *per MPI process* for each dimension. Each dimension is divided into as many parts as defined by *P*.Sizes are separated by a lower-case `x`, for example 2 `x` 2. If a dimension is given a size of 1 no distribution along that dimension is applied.
- `-print-defaults` print compiled-in defaults of all settings
- `-setparam:`*Q*=*v* initializes parameter *Q* with value *v*. The parameters that can be set here are those that have been specified in the CAOS source code using the `param` keywork. A list of all compiled-in parameters is displayed within the help text of the binary (`-help` option).

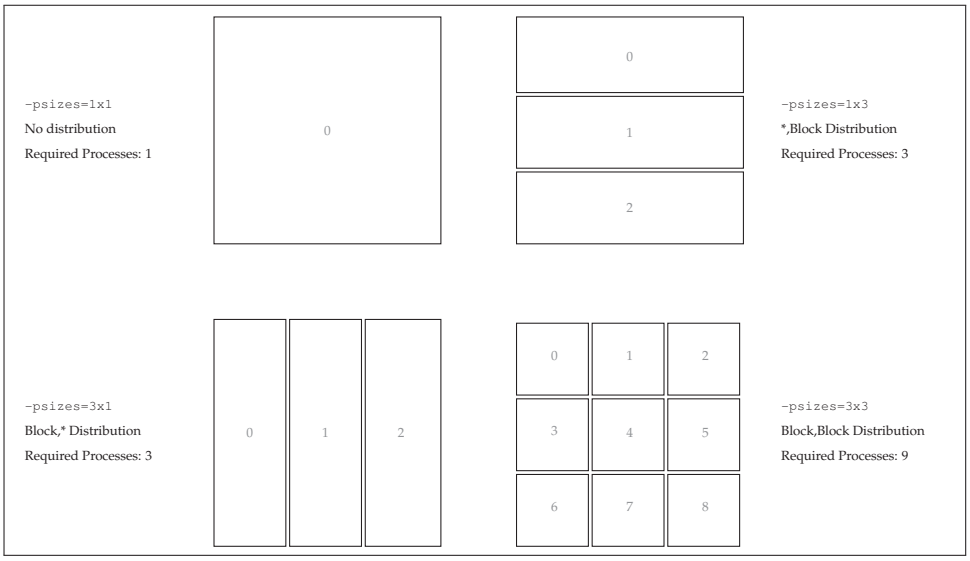

Fig. 17. Examples of possible distributions of a two-dimensional grid using the `-psizes` parameter

For a more detailed treatment of the semantics of these parameters and for an in-depth discussion of all implementation details, please see Grelck & Penczek (2007).

4. Selected implementation aspects

The main component of the CAOS tool-chain is the compiler that infers all static information of a CAOS program and compiles many default settings into the binary file. Still, certain

dynamic aspects of a simulation are determined at runtime when a simulation is run with values other than the compile-time defaults. For parallel execution, this includes the distribution of the cell grid and appropriate communication patterns. An overview of the general execution of a parallel CAOS program is shown in Fig. 18. We will describe a few

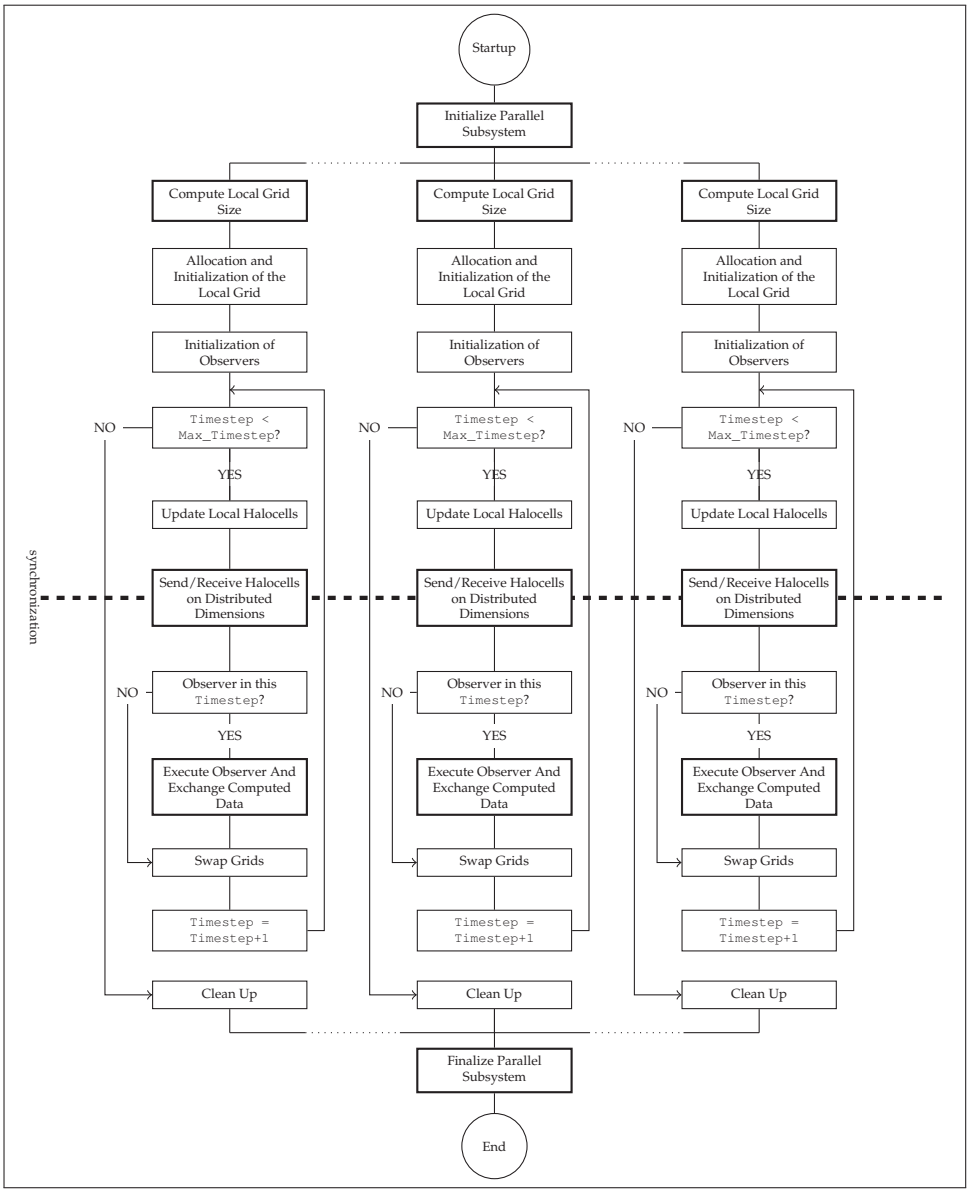

Fig. 18. Program flow plan of a parallel CAOS program

selected aspects of this process in more detail in the following sections.

4.1 Halo-cells: Inference, distribution and communication

In an implementation of a cellular automata simulation the cell grid is inevitably bounded in size, i.e. there will be certain cells that live on the boundary of the grid. If a dimension is defined as being cyclic, the neighbouring cell at the boundary lies at the other end of the grid. If the boundary is a static one the cells does not have a real neighbour. To avoid out-of-bounds errors without changing the access pattern of boundary cells the grid has to be padded as shown in Fig. 19 so that accesses to all neighbours can succeed.

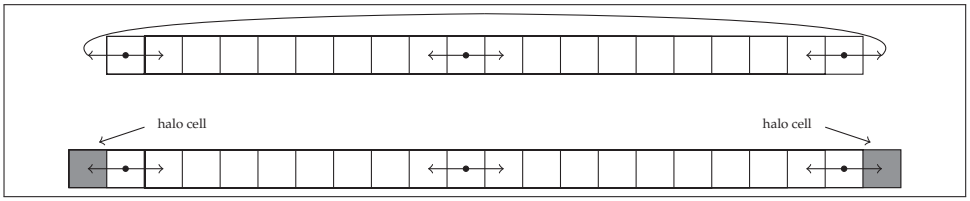

Fig. 19. Access to adjacent cells at cyclic (top) and static boundaries (bottom). At static boundaries halo-cells are automatically introduced.

The padding of the original cell grid introduces a frame of halo cells. This frame extends the original cell grid on each side by as many cells as are required for all accesses within a `behavior` to stay within the framed grid. An automatic inference mechanism analyses access patterns and determines the minimum size of the halo frame. Generally, this will lead to different extensions of dimensions as shown in Fig. 20. Halo cells are built from the same

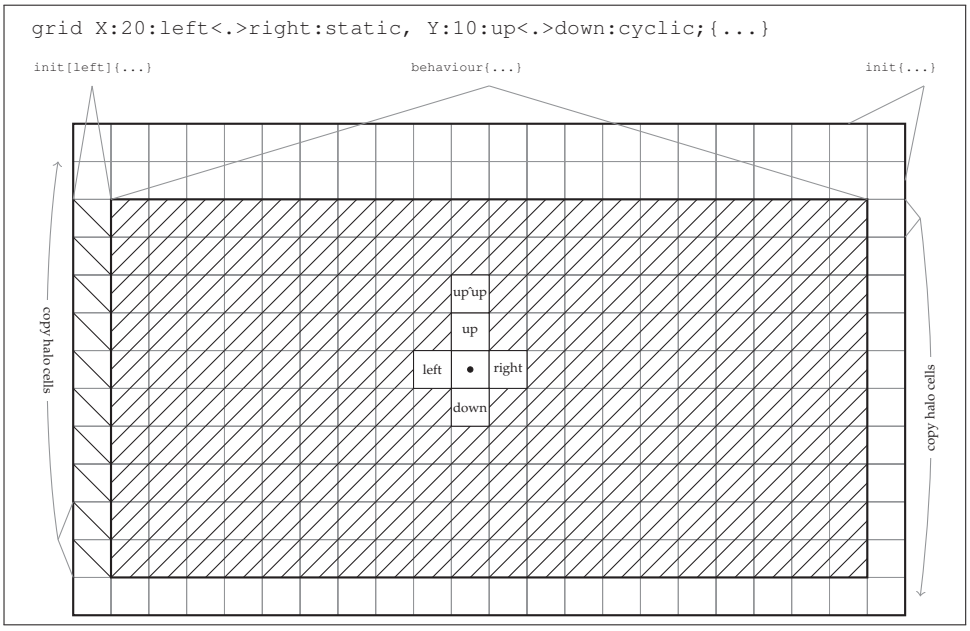

Fig. 20. Automatic embedding of the cell grid into a frame of halo cells. The extend of the frame is automatically inferred.

constituents as standard cells as defined in the `cell` block of a CAOS program. Consequently,

the initial values of halo cells are set by the `init` block. However, if more control over these initial values is required, `init` blocks for all or some of the halo frames may be given in additional `init` blocks. Fig. 20 shows this for a two-dimensional grid where the left side of the halo frame is initialised with special values.
Halo cells do not perform state transitions, they remain static during the entire simulation. If a grid dimension is defined as being cyclic, however, the halo cells are automatically updated after each time step so that cells on boundaries access the correct values of cells that are located on the other end of the grid. Because of the halo cell frame, these accesses are just applications of the standard access pattern and do not require complex index transformations.
In a parallel setting, where the global grid is divided up into several smaller grids, the halo frames extend each local grid. Along dimensions that are not distributed the halo cells serve the same purpose as before. Along dimensions that are distributed, however, the halo cells are used to represent cells of neighbouring local grids. This ensures that distribution is completely transparent for the rest of the implementation, especially the implementation of the state transition function. During each time step of the simulation the values of halo cells are exchanged with the appropriate adjacent neighbour of a distributed dimension as shown on Fig. 21.

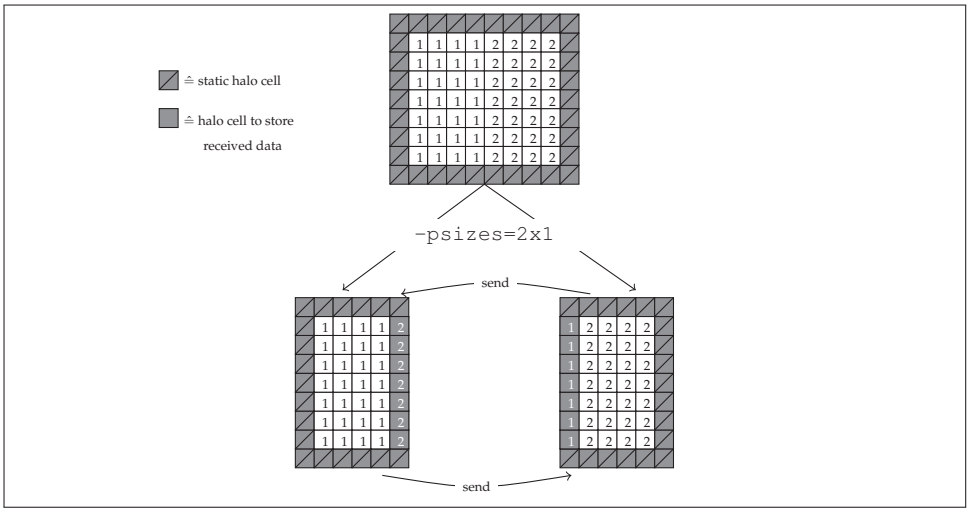

Fig. 21. Halo cells serve a dual purpose on a distributed grid. Along distributed axises the halo frame keeps copies of boundary cells of neighbouring local grids.

4.2 Parallel I/O

Writing data during the execution of observers poses a challenge when a simulation is executed in parallel. As the global grid does not exist as a whole, but is distributed over several processes, several concurrent write operations have to be carried out to the same file. Of course, this could be avoided by collecting all local grids in one process and then executing the observer code on a completely reassembled grid. Obviously, this procedure requires a considerable amount of data to be send and it may also require more memory than a single process has available to store the grid. The CAOS implementation takes a different approach where each process writes its local grid directly to the appropriate location within a shared file.

Each process uses information about the extent of the global grid and the distribution of the grid across several processes. From this information a process computes the location of the part of the global grid that it locally holds. This determines which regions of a file the process will fill with the contents of its local grid during the execution of an observer.
The MPI standard defines a set of I/O functions that offers high-performance I/O operations in a distributed setting on a high level of abstractions. For CAOS we make use of the *file view* concept Gropp et al. (1999), which uses information about the global grid and the information about the distribution to organise concurrent access to files into independent tasks. The file view determines which (not necessarily consecutive) parts of a file are visible to a process. The remaining parts of the file that belong to other processes are masked out. Using this, each process may apply its `observe` blocks in the same way as for sequential execution as the MPI I/O functions automatically direct file access to the appropriate location of the observer file. See Fig. 22 for an illustration.

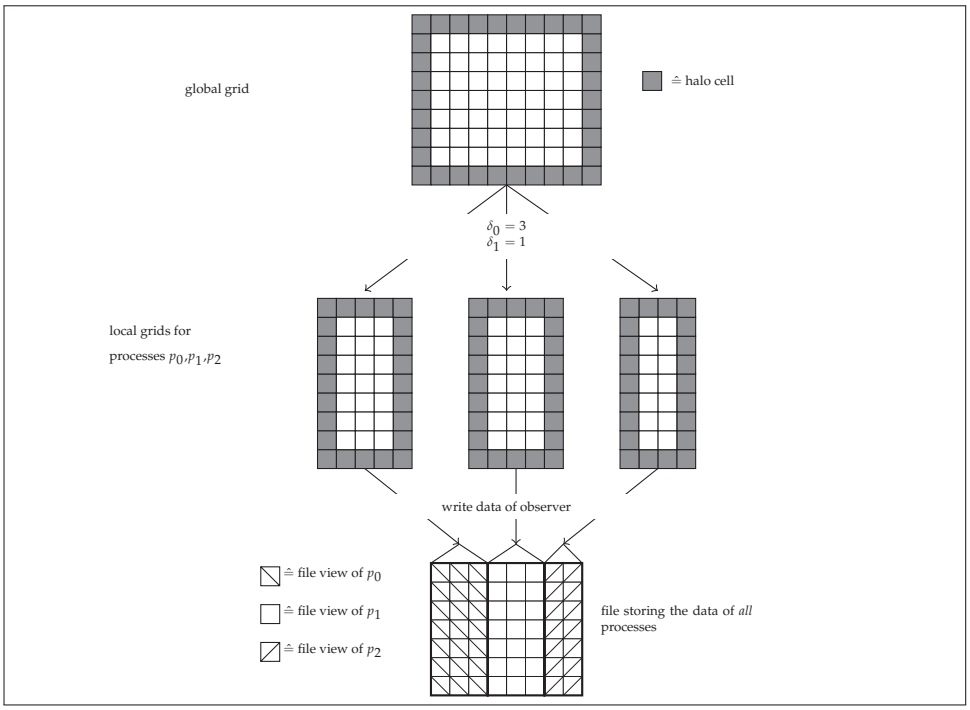

Fig. 22. Using MPI's dedicated high-performance I/O functionality, local *file views* are created for each local grid.

5. Evaluation and performance

We use an implementation of a 2-dimensional Jacobi iteration as the basis for performance evaluation experiments. Fig. 23 shows the complete CAOS code, which also serves as a reference example for CAOS programs.

We measured the runtime of this program on several machines to cover multiple common hardware configurations: a dual-core laptop, a cluster of distributed-memory blade servers and a 48-core shared-memory computation server.

```
param int atTstep = 1;
param int size = 100;

grid 0..size : left <.> right : static,
     0..size :   up <.> down  : static;

cells { double state; }

init { state = 0.0; }

init[down] { state = 500.0; }

behaviour { double a = 0.0;
            foreach (dir d in [up,down,left,right]) {
              a = a + state[d];
            }
            state = a / 4.0;
          }

observeall ("jacobi.outfile.all", timestep==atTstep) {
  double "state" = state;
}

observe ("jacobi.outfile.reduce", timestep==atTstep) {
  double "avgState" = avg(state);
}
```

Fig. 23. Jacobi iteration specified in CAOS

5.1 Performance on consumer-grade hardware

As a representative measurement for consumer-grade hardware, we have measured the Jacobi iteration on a 4096×4096 grid for 1000 timesteps. The laptop runs Mac OS X 10.6.4 on a 2.4GHz Intel Core 2 Duo and contains 4GB of main memory. The simulation has been compiled using the OpenMP back-end of the CAOS compiler. The generated code exploited both cores when run with two threads and achieved a speed-up of almost 1.8 as Fig. 24 shows.

5.2 Performance on distributed memory

The cluster consists of nodes with 2 E5520 Intel Xeon processors with hyperthreading disabled. Each node contains 24GB of ram, network connections between the nodes are established via DDR Infiniband. All nodes have access to a shared file system.
The runtimes shown in Fig. 25 are based on the execution of the Jacobi iteration on a 16384×16384 grid for 1000 time steps. Although the effect diminishes with increasing numbers of MPI processes, the simulation runtimes decrease with the number of available computing resources.

5.3 Performance on shared memory

The machine we have used for these runs is a 48-core shared-memory system consisting of 4 twelve-core AMD Opteron 6174 processors. The total amount of memory in the system is 256GB to which the cores have non-uniform access.
We ran two series of experiments, both using the Jacobi implementation on a grid of 16384×16384 for 1000 time steps. The first series of experiments used the OpenMP back-end of the CAOS compiler, the second series was distributed using MPI. As Fig. 26 shows, both versions

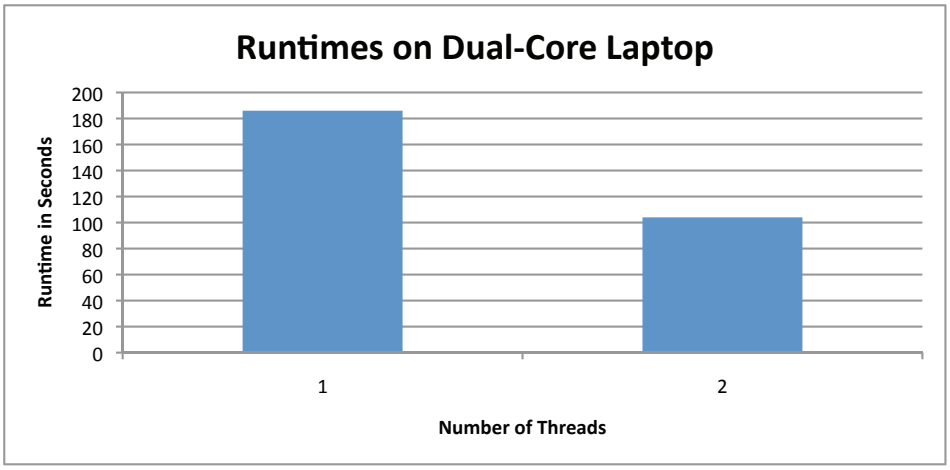

Fig. 24. Runtimes on a standard off-the-shelf laptop.

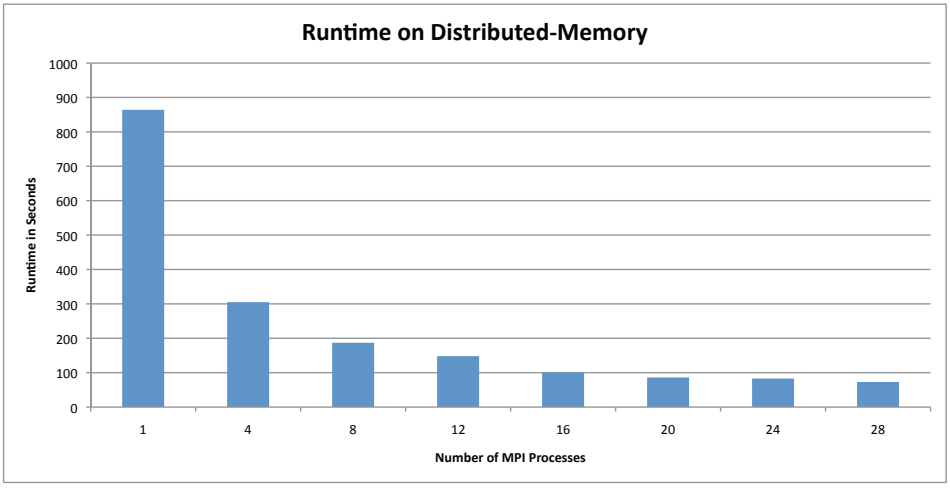

Fig. 25. Runtimes on a distributed-memory cluster using MPI

scale for a small number of cores. The OpenMP version is more efficient than the MPI variant for a small numbers of cores. However, on this machine the OpenMP implementation did not scale beyond 6 cores for reasons that would require further investigation. The MPI version did not suffer from this problem and scaled with the number of cores which reduced the runtime of the simulation considerably.

6. Related work

Mathematica and MatLab are well-known general-purpose systems that are also suitable for implementing cellular automata on a level of abstraction that exceeds that of standard

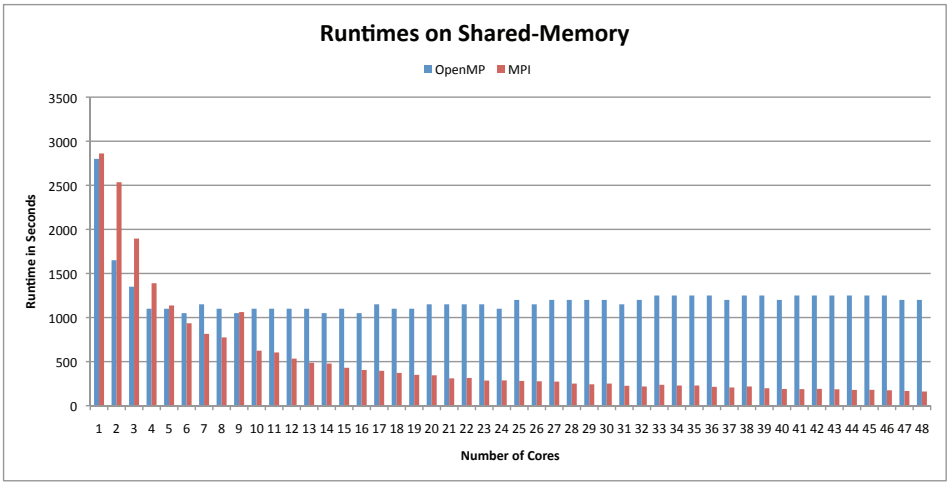

Fig. 26. Runtimes on a 48-core shared-memory machine using MPI and OpenMPI

programming languages. However, when it comes to complex simulations it is often more convenient to use a domain-specific, high-level language to implement the simulation. A programmer may choose from a range of languages like CANL Calidonna & Furnari (2004), CDL Hochberger et al. (1995), TREND Chou et al. (2002) and JCASim Freiwald & Weimar (2002). These languages offer an instruction set that is specifically tailored towards cellular automata. The simulations, however, are restricted to two-dimensional automata. Languages like CARPET/CAMEL Spezzano & Talia (1997a;b) and CELLANG Eckart (1992) overcome this restriction and offer support for automata with one, two and three dimensions. CAOS takes the idea even further and supports an arbitrary amount of dimensions.

The parallel execution of simulations is for example supported by CARPET Spezzano & Talia (1997a) with the CAMEL system Naumov (2004). CAOS supports the multi-threaded execution on shared memory machines and a distributed execution on clusters. If cluster nodes are multi-processor, shared-memory machines, CAOS also supports the combination of both models. Processes are distributed over the cluster nodes where the execution of each process is multi-threaded.

The regular structure of grids of cellular automata makes them well-suited for a direct mapping onto reconfigurable hardware. In Halbach et al. (2004) the simulation of cellular automata on FPGAs is investigated; in Ackermann et al. (2001) the design of a pseudo-random number generator in hardware on the basis of a CA is described. Compilers that translate cellular automata programs to these platforms are available, as for example the CDL compiler Hochberger et al. (1995).

7. Conclusion

CAOS is a new domain-specific programming language for the high-level specification of numerical simulations based on the well-known concept of cellular automata. CAOS extends this concept in a number of directions. For instance, grids are not limited to vectors or matrices, but can actually have any number of dimensions/axes. Communication is not restricted to nearest neighbours, but may cover any (static) neighbourhood. Cells do not carry binary information, but aggregate any number of numerical properties or attributes

as required by the programmer. Last not least, the state transition function is defined by means of a simple but nevertheless fully-fledged programming language whose design is geared towards the given purpose and, in particular, features a number of non-deterministic constructs.

Our CAOS compiler exploits the restricted pattern of communication characteristic for cellular automata for generating executable code whose runtime performance is highly competitive on modern computer architectures. Fully automatic parallelisation for shared memory architectures based on OpenMP as well as for distributed memory architectures based on MPI provides easy access to high-performance computing infrastructures from state-of-the-art symmetric multiprocessors to clusters of workstations and supercomputers. All this can be harnessed with only modest (sequential) programming skills and practically no familiarity with modern computer architecture or parallel computing issues.

We currently pursue two directions of future work. Firstly, we plan to continue on the successful route to support compiler-directed parallelisation through a restricted model of computation and extend the CAOS compiler to support emerging architectures such as general purpose graphics processors. Secondly, we are working on completing the CAOS language by support for agents that substantially increase the expressiveness of CAOS for advanced simulation.

Further information on the CAOS project, including a technical report that covers compilation in-depth Grelck & Penczek (2007) and a source distribution with demos for download, is available at

`http://www.caos-home.org/` .

8. References

Ackermann, J., Tangen, U., Bödekker, B., Breyer, J., Stoll, E. & McCaskill, J. (2001). Parallel random number generator for inexpensive configurable hardware cells, *Computer Physics Communications* 140: 293–302.

Amarasinghe, S. (2008). (How) can Programmers Conquer the Multicore Menace?, *17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, ACM, New York, NY, USA, pp. 133–133.

Boccara, N., Goles, E., Martinez, S. & Picco, P. (eds) (1993). *Cryptography with Dynamical Systems*, Kluwer Academic Publishers, pp. 237–274.

Calidonna, C. & Furnari, M. (2004). The cellular automata network compiler system: Modules and features, *International Conference on Parallel Computing in Electrical Engineering*, pp. 271–276.

Canyurt, O. & Hajela, P. (2005). A cellular framework for structural analysis and optimization, *Computer Methods in Applied Mechanics and Engineering* 194: 3516–3534.

Chapman, B. (2007). The Multicore Programming Challenge, *Advanced Parallel Processing Technologies*, p. 3.

Chou, H., Huang, W. & Reggia, J. A. (2002). The Trend cellular automata programming environment, *SIMULATION* 78: 59–75.

Conway, J. (1970). The game of life, *Scientific American* .

Dagum, L. & Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming, *IEEE Transactions on Computational Science and Engineering* 5(1).

D'Ambrosio, D., Iovine, G., Spataro, W. & Miyamoto, H. (2007). A macroscopic collisional model for debris-flows simulation, *Environmental Modelling & Software* 22: 1417–1436.

Eckart, D. (1992). A cellular automata simulation system: Version 2.0, *ACM SIGPLAN Notices* 27.

Ermentrout, G. B. & Edelstein-Keshet, L. (1993). Cellular automata approaches to biological modeling, *Journal of Theoretical Biology* 160: 97–133.

Freiwald, U. & Weimar, J. (2002). The Java based cellular automata simulation system JCASim, *Future Generation Computing Systems* 18: 995–1004.

Gabb, H., Mattson, T. & Breshears, C. (2009). Thinking in Parallel - Three engineers' Viewpoints, *Intel Software Insight Magazine* 16: 24–26.

Georgoudas, I. G., Sirakoulis, G. C., Scordilis, E. M. & Andreadis, I. (2007). A cellular automaton simulation tool for modelling seismicity in the region of Xanthi, *Environmental Modelling & Software* 22(6): 1455–1464.

Grelck, C. & Penczek, F. (2007). CAOS: A Domain-Specific Language for the Parallel Simulation of Extended Cellular Automata and its Implementation, *Technical report*, University of Lübeck, Institute of Software Technology and Programming Languages.

Gropp, W., Lusk, E. & Skjellum, A. (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, Massachusetts, USA.

Gropp, W., Lusk, E. & Thakur, R. (1999). *Using MPI-2*, The MIT Press, chapter Parallel I/O, pp. 60–64.

Guisado, J., de Vega, F. F., Jiménez-Morales, F. & Iskra, K. (2006). Parallel implementation of a cellular automaton model for the simulation of laser dynamics, *LNCS* 3993: 281–288.

Halbach, M., Hoffmann, R. & Röder, P. (2004). FPGA implementation of cellular automata compared to software implementation, *Organic and Pervasive Computing*, Lecture Notes in Informatics, pp. 309–317.

Hochberger, C., Hoffmann, R. & Waldschmidt, S. (1995). Compilation of CDL for different target architectures, *Parallel Computing Technologies*, Vol. 964 of *LNCS*, Springer, pp. 169–179.

Nagel, K. & Schreckenberg, M. (1992). A cellular automaton model for freeway traffic, *J. Phys. I France 2* .

Naumov, L. (2004). CAME&L – cellular automata modeling environment and library, *LNCS* 3305: 735–744.

Popovici, A. & Popovici, D. (2002). Cellular automata in image processing, *Technical report*, University of the West Timisoara, Romania.

Spezzano, G. & Talia, D. (1997a). A high-level cellular programming model for massively parallel processing, *Proc. 2nd Int. Workshop on High-Level Programming Models and Supportive Environments (HIPS'97)*, IEEE Press, pp. 55–63.

Spezzano, G. & Talia, D. (1997b). Programming high performance models of soil contamination by a cellular automata language, *High-Performance Computing and Networking*, Vol. 1225 of *LNCS*, Springer, pp. 531–540.

Stevens, D., Dragicevic, S. & Rothley, K. (2007). iCity: A GIS-CA modelling tool for urban planning and decision making, *Environmental Modelling & Software* 22(6): 761–773.

Cellular Automata - Simplicity Behind Complexity

Edited by Dr. Alejandro Salcido

ISBN 978-953-307-230-2
Hard cover, 566 pages
Publisher InTech
Published online 11, April, 2011
Published in print edition April, 2011

Cellular automata make up a class of completely discrete dynamical systems, which have became a core subject in the sciences of complexity due to their conceptual simplicity, easiness of implementation for computer simulation, and their ability to exhibit a wide variety of amazingly complex behavior. The feature of simplicity behind complexity of cellular automata has attracted the researchers' attention from a wide range of divergent fields of study of science, which extend from the exact disciplines of mathematical physics up to the social ones, and beyond. Numerous complex systems containing many discrete elements with local interactions have been and are being conveniently modelled as cellular automata. In this book, the versatility of cellular automata as models for a wide diversity of complex systems is underlined through the study of a number of outstanding problems using these innovative techniques for modelling and simulation.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821