CAOS: A Domain-Specific Language for the Parallel Simulation of Cellular Automata

Clemens Grelck[1,2], Frank Penczek[1,2], and Kai Trojahner[2]

[1] University of Hertfordshire
Department of Computer Science
c.grelck@herts.ac.uk, f.penczek@herts.ac.uk
[2] University of Lübeck
Institute of Software Technology and Programming Languages
trojahner@isp.uni-luebeck.de

Abstract. We present the design and implementation of CAOS, a domain-specific high-level programming language for the parallel simulation of extended cellular automata. CAOS allows scientists to specify complex simulations with limited programming skills and effort. Yet the CAOS compiler generates efficiently executable code that automatically harnesses the potential of contemporary multi-core processors, shared memory multiprocessors, workstation clusters and supercomputers.

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 [1,2,3,4,5,6,7,8]. 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 correctly and efficiently quickly turns into a painful venture distracting from the interesting aspects of the simulation problem itself. Current advances in computer architecture make the situation considerably worse. Abundance of parallel processing power through multicore technology and the need to parallelise simulation software to effectively use standard computing machinery confronts us with the notorious hazards of parallel programming. The model of cellular automata naturally lends itself to parallel execution. However, the effective utilisation of parallel processing resources on whatever level requires very specific programming skills and is difficult, time-consuming and error-prone.

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, it provides the scientist with little programming experience 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 and distributed

V. Malyshkin (Ed.): PaCT 2007, LNCS 4671, pp. 410–417, 2007.

memory systems through MPI. In fact, both approaches may be combined having the compiler generate multithreaded OPENMP code within MPI processes for hybrid architectures. Thus, CAOS not only supports today's multicore processors, but likewise clusters of workstations, traditional supercomputers and combinations thereof.

The remainder of the paper is organised as follows: In Sections 2, 3 and 4 we introduce cells, agents and observers, respectively. Section 5 outlines some implementation aspects and reports on performance measurements. We addresses related work in Section 6 and conclude in Section 7.

2 Cells

Fig. 1 shows the general layout of a CAOS program, which is organised into a sequence of sections. Following a set of declarations, which we will only sketch out briefly, we find the basic constituents of CAOS: cells, agents and observers. This section is concerned with cells; agents and observers are explained in the following sections. Cells consist of attribute declarations (the state space), an initialisation (the initial state) and a behaviour definition (the state transition function). This specification of a single cell is complemented by a grid definition that defines the assemblage of these uniform cells to a cellular automaton.

Program ⇒ *Declarations Cells Agents Observers*
Cells ⇒ *Grid Attributes Init Behaviour*
Grid ⇒ **grid** *Axis* ⌈ , *Axis* ⌋* ;
Axis ⇒ *Index* `..` *Index* : *Id* `<.>` *Id* : *Boundary*
Index ⇒ *IntConstant* | *Id*
Boundary ⇒ **static** | **cyclic**
Attributes ⇒ **cells** { ⌈*Type Id* ; ⌋+ }

Fig. 1. Grammar of CAOS programs

As mentioned earlier, the state space of CAOS cells can be quite complex: Following the key word `cells` we have a sequence of attribute declarations each associated with a type. This part very much resembles the definition of attributes in class definitions of an object-oriented languages. As types CAOS currently supports boolean values (`bool`), integer numbers (`int`), double precision floating point numbers (`double`) and user-defined enumeration types. The latter are very similar to those in C and can be defined in the declaration section of a CAOS program. Enumeration types are handy to use symbolic names whenever the state space is rather small. For example, in the Game of Life it may be more expressive to use an enumeration type

```
enum dead_or_alive {dead, alive};
```

than representing the state by boolean or integer values. The corresponding cell definition could look like

```
cells { dead_or_alive state; }
```

Cells are arranged to multi-dimensional grids using the grid declaration. Following the key word `grid` we have a sequence of axis specifications. Each axis specification itself consists of three parts separated by colons. First, we specify the extent of the grid along this axis. Grid sizes may be hard-coded using an integer constant. However, in most cases it is more convenient to use a symbolic constant defined in the declaration section or a symbolic parameter, which allows us to determine the grid size at runtime. A parameter declaration of the form

```
param int size = 100;
```

makes the CAOS compiler automatically generate a command line option `-size` *num* that can be used to overrule the default value specified (100 in this example). Of course, parameters can be used throughout the CAOS program at any appropriate expression position and not only in grid specifications.

The second part of an axis specification introduces two new identifiers as symbolic names for neighbouring cells along decreasing (left of `<.>` symbol) and increasing (right of `<.>` symbol) indices. These symbolic *directions* are the only means to access attributes of neighbouring cells; they avoid the error-prone use of explicit numerical indices and calculations on them.

Any grid has a finite size which raises the question of how to handle cells on the boundary. By putting one of the key words `static` and `cyclic` into the last part of the axis specification we offer the choice between an additional layer of constant cells and cyclic neighbourship relations. As an example, consider the following specification of a 2-dimensional grid using compass names for directions:

```
grid 1..100  : north <.> south : static,
     1..size : west  <.> east  : cyclic;
```

Cells may be initialised by the available set of constants and parameters. Furthermore, entire start configuration can be read from files. We skip this part of the language and head straight on to the more interesting behaviour specification, i.e. the state transition function of our cells. Fig. 2 defines the syntax. Essentially, a CAOS behaviour specification is a C- or Java-like block of assignments. In addition to the state identifiers declared in the cells section of a CAOS program, we have local variables in the behaviour section. Such local variables are pure placeholders for intermediate values. Apart from them, the body of a behaviour specification is a sequence of assignments to either local variables or state variables.

Behaviour ⇒ **behaviour** { [*Type Id* [= *Expr*] ;]* [*Instruction*]* }
Instruction ⇒ *Assignment* | *Cond* | *ForEach* | *Switch*
Assignment ⇒ *Id* = *Expr* ;
Cond ⇒ **if** (*Expr*) *Block* **else** *Block*
ForEach ⇒ **foreach** (*Type Id* **in** *Set*) *Block*
Switch ⇒ **switch** (*Id*) { [*Case*]+ [*Default*]
Case ⇒ **case** *CaseVal* [, *CaseVal*]* : *Block*
Block ⇒ *Instruction* | { [*Instruction*]* }

Fig. 2. Syntax of the behaviour section

Expressions are made up of local and state variables as well as the usual operators on the basic types supported by CAOS. The most noteworthy part here is the access to state variables of neighbouring cells. Whereas the existing values of a cell's own state variables are accessed simply by the variable's name, neighbouring cell's state variables are referred to using the *directions* introduced in the grid specification. Given the above cell and grid specifications, a cell may access the state of its left neighbour by `state[west]` or its upper neighbour by `state[north]`. CAOS also provides an associative and commutative operator on directions: With `state[north^west]` we can easily address the upper left neighbour without complicated and error-prone numerical index computations. Neighbourhoods are not limited to immediate neighbours: `state[north^north]` is perfectly fine. As the boundary condition on this axis was defined as static, the CAOS compiler will introduce a constant boundary layer of sufficient size.

CAOS provides a set of versatile control constructs. The C- or Java-like conditional, for example, allows us to implement the Game of Life:

```
behaviour { int cnt = 0;

            if (state[north]       == alive) cnt = cnt + 1;
            if (state[south]       == alive) cnt = cnt + 1;
            if (state[east]        == alive) cnt = cnt + 1;
            if (state[west]        == alive) cnt = cnt + 1;
            if (state[north^east]  == alive) cnt = cnt + 1;
            if (state[north^west]  == alive) cnt = cnt + 1;
            if (state[south^east]  == alive) cnt = cnt + 1;
            if (state[south^west]  == alive) cnt = cnt + 1;

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

A more concise specification of the counting process can be achieved using the `foreach` construct:

```
foreach (dir d in [north, south, east, west,
                   north^east, north^west, south^east, south^west]) {
  if (state[d] == alive) cnt = cnt + 1;
}
```

The body code is executed for each element of the set of directions, represented by the local variable `d` of type `dir`. Likewise, the decision making code may be written more intuitively thanks to the `switch`-construct:

```
switch (state) { case alive: switch (cnt) { case 2,3: state = alive;
                                              default:  state = dead;
                                            }
                 case dead:  switch (cnt) { case 3:   state = alive;
                                              default:  state = dead;
               }                          }
```

CAOS provides further variations of the `foreach` and `switch` constructs using explicit guard expression. Moreover, there is a range of probabilistic constructs that allow programmers to introduce non-determinism. However, due to the limited space we cannot elaborate on them.

3 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 neighbourship 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.

4 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.

5 Implementation and Evaluation

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 user-defined number of OpenMP threads, provided that the available Mpi implementation is thread-safe. Proper

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

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. For implementation details see [16].

We use 2-dimensional Jacobi iteration as the basis for some performance evaluating experiments. Fig. 3 shows the complete CAOS code, which also serves as a reference example for CAOS programs. Note the observers that, at a certain time step (usually the last one) save the entire state as well as the average. Our experiments use two different machines: a 72-processor SUN SF15k with NUMA shared address space and an 8-node PC cluster with Intel Pentium IV processors and a gigabit ethernet connection.

```
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. 3. Jacobi iteration specified in CAOS

Fig. 4 shows the outcomes of our experiments on the shared address space (left) and on the distributed memory (right) architecture. While the latter figures show good speedups and scalability in the range of available nodes, the former provide some interesting insights into the suitability of Mpi, OpenMP and a combination of the two as low-level execution models for CAOS (and similar numerical codes), at least on the given machinery. Using our purely Mpi-based code generator achieves substantially better performance values than the purely OpenMP-based one. Likewise, using 2 or 4 OpenMP threads inside each Mpi process does not pay off, although this organisation exactly matches the SF15k architecture. The SUN Mpi implementation seems to be considerably more advanced than the OpenMP support in the C compiler. We also assume that the

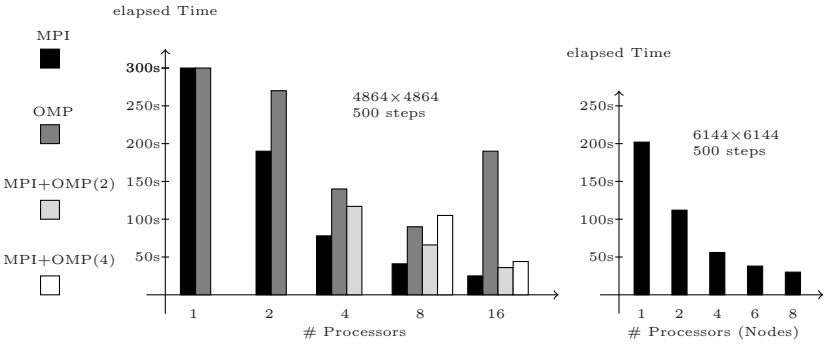

Fig. 4. Execution Times of CAOS Jacobi iteration on a shared address space system (left) using either Mpi or OpenMP or a combination of both as execution platform and on a distributed memory workstation cluster (right) using only Mpi

Solaris operating system may not schedule threads in a way that harnesses the hierarchical memory organisation.

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 programming languages. As examples for domain-specific languages and systems we mention CANL [9], CDL [10], TREND [11], CARPET/CAMEL [12,13], CELLANG [14] and JCASim [15]. Limited space does not allow us to discuss their relationship with CAOS in the desirable detail. A similar purpose inevitably results in certain similarities; differences lie in the number of axes supported, the concrete syntactical support for high-level programming, the way concurrency in the cellular automaton model is exploited (if at all) and the orientation towards runtime performance in general.

7 Conclusion

CAOS is a new domain-specific programming language that supports specification of multidimensional extended cellular automata at a high level of abstraction. Our automatically parallelising compiler exploits the restricted pattern of communication for generating efficiently executable code for both shared and distributed memory architectures. It provides access to the potential of modern computer architectures with modest programming skills. Space limitations prevent us from giving a complete introduction to the CAOS language. Further information on the project, including a technical report that covers compilation in-depth [16] and a source distribution with demos for download, is available at `http://caos.isp.uni-luebeck.de/`

References

1. Ermentrout, G.B., Edelstein-Keshet, L.: Cellular automata approaches to biological modeling. Journal of Theoretical Biology 160, 97–133 (1993)
2. Gutowitz, H.: Cryptography with Dynamical Systems, pp. 237–274. Kluwer Academic Publishers, Boston (1993)
3. Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. J. Phys. I France 2 (1992)
4. Guisado, J., de Vega, F.F., Jiménez-Morales, F., Iskra, K.: Parallel implementation of a cellular automaton model for the simulation of laser dynamics. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2006. LNCS, vol. 3993, pp. 281–288. Springer, Heidelberg (2006)
5. Stevens, D., Dragicevic, S., Rothley, K.: iCity: A GIS-CA modelling tool for urban planning and decision making. Environmental Modelling & Software 22 (2007)
6. Georgoudas, I.G., Sirakoulis, G.C., Scordilis, E.M., Andreadis, I.: A cellular automaton simulation tool for modelling seismicity in the region of Xanthi. Environmental Modelling & Software 22 (2007)
7. D'Ambrosio, D., Iovine, G., Spataro, W., Miyamoto, H.: A macroscopic collisional model for debris-flows simulation. Environmental Modelling & Software 22 (2007)
8. Canyurt, O., Hajela, P.: A cellular framework for structural analysis and optimization. Computer Methods in Applied Mechanics and Engineering 194 (2005)
9. Calidonna, C., Furnari, M.: The cellular automata network compiler system: Modules and features. In: International Conference on Parallel Computing in Electrical Engineering, pp. 271–276 (2004)
10. Hochberger, C., Hoffmann, R., Waldschmidt, S.: Compilation of CDL for different target architectures. In: Malyshkin, V. (ed.) Parallel Computing Technologies. LNCS, vol. 964, pp. 169–179. Springer, Heidelberg (1995)
11. Chou, H., Huang, W., Reggia, J.A.: The Trend cellular automata programming environment. SIMULATION 78, 59–75 (2002)
12. Spezzano, G., Talia, D.: A high-level cellular programming model for massively parallel processing. In: Proc. 2nd Int. Workshop on High-Level Programming Models and Supportive Environments (HIPS'97), pp. 55–63. IEEE Press, New York (1997)
13. Spezzano, G., Talia, D.: Programming high performance models of soil contamination by a cellular automata language. In: Hertzberger, B., Sloot, P.M.A. (eds.) High-Performance Computing and Networking. LNCS, vol. 1225, pp. 531–540. Springer, Heidelberg (1997)
14. Eckart, D.: A cellular automata simulation system: Version 2.0. ACM SIGPLAN Notices 27 (1992)
15. Freiwald, U., Weimar, J.: The Java based cellular automata simulation system JCASim. Future Generation Computing Systems 18, 995–1004 (2002)
16. Grelck, C., Penczek, F.: 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 (2007)