The Cellular Automata Network Compiler System: modules and features

C.R. Calidonna, M. Mango Furnari
Istituto di Cibernetica"E. Caianiello
Istituto di Cibernetica"E. Caianiello" C.N.R
Viale Campi Flegrei 34 Pozzuoli (NA) Italy
{c.calidonna,mf @cib.na.cnr.it

Abstract

In simulating complex phenomena by Cellular Automata (CA) programming model performances assume a crucial role. The efficient exploitation of intrinsic parallelism is a must for this kind of applications. The Cellular Automata Network (CAN) model, that is an extension of the CA classical model due to the introduction of the network abstraction between automata nodes, offer another parallelism source besides to the classical CA parallelism source. In order to exploit efficiently this opportunities tools and systems must be designed. In this paper we deal with the CAN compiler system components description and features designed ad hoc.

1. Introduction

The Cellular Automata (CA) [9,10] programming model offers an intrinsic data parallelism sources. CA is one of the main successful computational paradigms in simulating complex systems characterized by discrete spatio-temporal phenomenological properties.

A cellular automaton consists of: a regular uniform lattice with a discrete variable at each site (cell) whose variables are updated simultaneously (synchronously), based on the values of the variables in their neighborhood at the preceding time step, and according to a definite set of “local rules” (named the transition function). The simultaneously variable cell’s updating addresses naturally data parallelism. Here, we use an extended cellular automaton computation model, based on the notion of network of cellular automata to simulate complex physical phenomena represented in terms of connected components (cellular automata nodes). At this purpose we use the high level CANL language [2] to express this model in of cellular automata and their composition in terms of network of cellular automata.

This approach allows the exploitation of two kinds of parallelism: one is the control parallelism coming from the possibility to execute in parallel more CA belonging to the network; the other one is the data parallelism intrinsic to the CA model. In order to manage the two parallel forms, eventually available, in a program written according to the CANL language a compiler, i.e. the CAN compiler, was built.

Here we describe the main characteristics of the designed compiler system. The paper is organized as follows in Section 2 we describe the CAN model and primitives of the language CANL [2] built ad hoc for the simulation of complex systems characterized by spatio-temporal discrete evolution. In Section 3 we deal with CAN compiler feature and components description. In Section 4 an example of expected output is described introducing a more detailed description of the scheduler module. Finally some conclusions and future works are given.

2. CAN Model and Language Primitives

The CAN model extends the standard CA model introducing the possibility to have a network of cellular automata, where each automaton represents a component of a physical system and connections among network automata represent a disjoinable evolutive laws that characterize the physical system to be simulated.

The CAN model allows the construction of complex physical phenomenon models, characterized by a two-level evolutionary process, deriving complex model components and their interactions by means of a process reduction: the model evolves on a basis of local cellular interaction rules together with automata (model components) connections.

In CAN model an automaton is denoted by a name, and its behavior is described by a set of properties (lattice cells types), by a transition function, and by a neighborhood type. A property corresponds either to a physical property of the system to be simulated such as temperature, volume and so on, or to some other feature

1 For brevity from this moment we will use the word automaton instead cellular automaton.
2 A micro transition for the cellular automaton transition and a macro transition for the entire complex system transition.
of the system such as the probability of a particle to move and so on.

In this schema, a cell of an automaton is considered as a composition of the cells of the automaton itself properties

The introduction of several complex system components, through the network of cellular automata abstraction, implies the individuation of corresponding automata where each of them has its own properties.

According to the CAN model the ownership of properties implies due that when a component/automaton requires to know the values of the cells of another property, in order to evolve, it means that these components are partially coupled: this case in which it is necessary to use a network of cellular automata.

From an execution point of view the partial coupling between components introduce a precedence execution order between them: if a property $P_1$ needs to know the cell values of the property $P_2$ to evolve at each time step, a network of two automata, $A$ and $B$, have to be defined each one having respectively the property $P_1$ and $P_2$, where $A$ is the owner of the property $P_1$, and $B$ is the owner of the property $P_2$.

The possibility to express, in CAN model, the different components of a complex system in terms of CA allows also for the exploitation of another source of parallelism that can improve application performances; it is called task parallelism coming from the possibility to concurrently execute network automata

Now let us describe a CAN program introducing the primitive the language; a CANL program is essentially a set of declarations where the execution order is state on the base of network relation declaration that can relax the lexicographic order in which the automata declarations appear, as sketched in the figure above (cfr. Fig. 1)

Let us resume the main language primitives in CANL as shown above together with a program declaration.

First of all the user responsible to write and express the dependence relations between automata variable.

The principal primitives of the CANL language are: def-net used to express precedence relation between automata and the root graph, as the network is represented as an acyclic graph for each time step, has not precedence; def-automaton used to define a network automaton; is univocally identified by its name and a corresponding transition function def-transition name and properties names.

The transition function acts on automaton properties in order to determine, at each simulation step, the state of the automaton. It is possible to define synchronous and asynchronous transition functions.

The transition function body actually uses the form and syntax of an ordinary functional-like language, and it is built around a set of primitives and user defined functions called the auxiliary functions. The auxiliary functions are user-defined functions where recursion is also available. The topology specifies the neighborhood type to be used in computing the transition function. Currently the available topologies are: North East West South (NEWS).

Figure 1 CAN Program and primitives
In the CANL language some collective operators are available: **set-all row** (**set-all-column**) used to assign the value to all the elements of the raw i (column j); **set-rest** used to assign a value to all the unassigned grid cells.

The function **initialize** used to initialize the properties’ grids. In an automata network, sometimes it is necessary to have a control construct in order to apply the transition function repeatedly until a condition is verified, where he number of times the transition function is applied on the same automaton depends on a given condition. In order to execute the transition: the **execute-transition-function-again-if** (**etfai**) control construct is introduced in the CANL language.

3. CAN Compiler modules

In the past many languages and environments for CA based model simulation were proposed; a first extensive review can be found in Worsch [10]. Many of the implemented systems, as parallel complex system simulation tools, propose similar techniques and components as the underlying basic concepts and core aspects are the same. Environments as CAMEL [4] or PECANS [3] are mainly specialized in model specification, systems as P-CAM [8] stress parallel computers point of view using optimally parallel computing resources. Tools as JCA Si m [5] focused on portability implementing the system entirely in Java. Systems as NEMO [6], mainly applied to geographic Information Systems (GIS), offers a high degree of transparent parallelism coping with load balancing techniques. The most part of this system, when specialized in parallelism management, cope with load balancing techniques, ours is the case in which mainly scheduling strategy should be addressed as several concurrent parallelism opportunities, when available, must be managed. Eventually also load balancing techniques are necessary, but this will be our next work.

Starting from the existing language CANL [2], we thought to take into account together the following aspects: model specification, portability and parallelism exploitation in the designing activity of the CAN compiler.

The choice of designing such a compiler system was driven in order to store and capture some specific language features and control mechanism and furthermore to exploit and express intrinsic parallelism opportunities.

The designed CANL compiler has two main modules: a **Front-End** and a **Back-End**.

The Front-End is characterized by: syntactical analysis and semantic check to assure automata properties ownership in order to avoid write operations by means of other automata.

Two main data structures, together with their manipulation functions, must be available since the parsing action:

1. the **Abstract Syntax Tree** (**AST**) that must be built around main syntactical categories:
   a) **AST_automata_network** in order to store any information/annotation on the automata network structure
   b) **AST_automaton** in order to store any information/annotation on the automata structure.

2. the **Symbol Table** (**ST**) that must register any symbol in CANL programs, expressing also their scopes and extensions.
It is worth of note that informations and annotations, collected on the program during parsing phase, and stored in the AST and ST data structures are not suitable to be used directly for code optimization. Other relation and properties must be discovery on the code. Intermediate Representations (IR), such as Control Flow Graph (CFG) or Call Graph (CG) [1] are universal representation that allow for capturing main properties on program control flows and dependencies. For this reason we decided to design and introduce a module in which two main Intermediate Representation (IR) are built and manipulates. The considered IR are respectively the:

1) Call Graph that takes into account the call graph for the transition functions, auxiliary functions and initialize function;
2) Control Flow Graph (CFG) for building the CFG of a CANL program and to prepare the system for an eventual application of interprocedural analysis algorithms.

The last module is the Code Generator (CG). On the base of existing IR and annotated information that translate a CANL syntax expressed program into a corresponding C syntax program that call the interface run-time functions for a suitable execution.

The back-end system (or run-time system) is further decomposed in a:

a) memory manager for data structure and function initialisation, their allocation and accesses functionalities. The module manages all needed informations regarding the memory space and techniques for read and write accesses blocks.

b) executor that initialises the execution environment according to collected information on intermediate representations. The module have the rule to initialise the thread environment also taking into account the number of thread according to a user suggestion, a memory module suggestions on data locality preservation or according a default number.

c) scheduler, that is essentially a dispatcher, generates the multitasking units, that can result in a multigrain set of tasks according to available parallelism. The scheduled task are managed according to the precedence relations annotated into the automataNet data structure.

The developed system due to modular design is easily expansible and other run-time systems can be added.

4. The CAN compiler run-time system: planning a CANL program execution

The run-time system takes into charge the management of the execution of a CANL program, as a CANL program essentially is a set of declaration as sketched in fig. 1.

An API, between the front-end and back-end, opportunely equipped with function prototypes, and management functions, was designed. Each module, in the run-time system, communicates with the others by mean of this interface: this choice was driven by modularity principle in order to be easily modifiable and eventually replaced in future releases without affecting other modules.

For portability reason we decided to use Pthread Library [7] exploiting available parallelism at a more high level.

In our approach we exploit parallelism at data parallel level, when available, in such a way when multiple nodes can be concurrently executed multigrain tasks can be derived.

Bulks of thread can be generated, gangs if they belong to the same transition function, respecting the precedence relation and synchronised when it is necessary.

As an example if we have an acyclic task graph as sketched in the figure below (fig. 3.a):

---

3 The multitasking available results in a coarse grain task if the transition function is asynchrony; i.e. the execution must be sequential, or in a thin grain task is data parallelism is available to a
the correspondent bulks of task executions are represented in the boxes of the figure 3.b. and each reaching arrow, representing the precedence relation, is a transferring control flux after a synchronisation point. Any task in each box can be further subdivided into subtask of a well defined granularity according to the stated number of stated threads.

The scheduler module takes into account the task graph and create the lists for each bulk of tasks; at this purpose two queues are planned: Queue_noprec in cui tasks
without precedence are present, and Queue_proc to take into account the processed tasks.
According to the CAN compiler schema (fig. 2) a C coded program could be one of the output of the CAN compiler system.
An example of translated code is shown in the figure 4 where a CANL program, whose model is taken from Toffoli [9].
On the left of the figure, in which two components are available, a CANL program is expressed and its translation into C language is shown in the right of the figure.
A translated CANL program, as sketched, is a set of:
1. define and includes statements;
2. declarations of global structures and/or global variable;
3. declarations of function prototypes and functions defined in the program (auxiliary functions).
4. a main program that drive the sequential execution of the program.

5. Conclusion and future work

The CANL compiler system here was presented as the first prototype of a system for parallelism opportunities annotations and discoveries on the base of the specific parallel language developed.
The driving modular principle for compiler design showed the flexibility of the run-time component.
In a near future we plan to adopt a run-time system for distributed memory architecture schema.
In this case more considerations and better solution must be addressed in order to study and solve load balancing problems, that in this case becomes crucial. Eventually migrations tasks policies will be studied and considered.

6. References