The ParCeL-2 Programming Language*

Paul-Jean Cagnard

Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland
cagnard@di.epfl.ch

Abstract. A semi-synchronous parallel programming language and the corresponding computing model are presented. The language is primarily designed for massively parallel fine grained applications such as cellular automata, finite element methods, partial differential equations or systolic algorithms. In this language it is assumed that the number of parallel processes in a program is much larger than the number of processors of the machine on which it is to be run. The computational model and the communication model of the language are presented. Finally, the language syntax and some performance measurements are presented.

1 Introduction

Many parallel programming models have been proposed in the past years as alternatives to both message passing and data parallel programming. Among the most prolific class of alternate models there is the family of languages based on concurrent objects or actors [1, 2].

In actor- or object-based models, parallelism of execution is expressed explicitly. These models allow to express the distribution of data explicitly and in a natural way. This opens interesting opportunities for automated data- and process-mapping.

Nearly all models based on concurrent objects or actors use asynchronous execution models, i.e., execution schemes where computation and message sending are not inherently automatically globally synchronised. This entails that communication is not easy to implement efficiently and concurrent object systems are often restricted to coarse grain parallelism for performance reasons.

In the following a parallel programming language whose computation model can be seen as an extension of the BSP [3] model and other synchronous object-oriented models is presented.

2 The ParCeL-2 Programming Model

A ParCeL-2, program is typically composed of a large number of fine grained processes executing concurrently. Each process execution consists of a sequence of supersteps. A superstep is composed of two phases: a computation phase and a communication phase. Supersteps are separated by synchronisation barriers.

* Work supported by the Swiss National Science Foundation under grant 21-49519.96.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 767–770, 2000.

During the computation phase of a superstep, a process executes computations which only manipulate data local to this process. A process can send data to other processes in the course of a computation phase. The effective transmission of data between processes happens at the end of each superstep. This means that no data are exchanged during computation phases. It can be observed that this computing model is an intrinsically distributed memory MIMD model.

Aggregates of processes can be constructed in order to build abstract processes whose behaviour is more complex than an elementary process but the rest of the program does not need to know how this behaviour is implemented. Aggregates of processes offer means of creating arbitrarily complex process topologies and hierarchical parallel program designs.

In ParCeL-2, unlike in BSP, every process, or cell, can execute supersteps with a slower frequency than that of the execution environment's global clock. This is why ParCeL-2 is called semi-synchronous. This is useful since complex cells containing other cells may need more time to produce a meaningful result than an elementary cell. This is a way of saying that if a cell has a synchronisation period n, data will only send along or read from its channels every n elementary supersteps.

In ParCeL-2, processes can only communicate through predeclared channels. Channels have four main properties. They are typed: a channel, like a local variable, is of a certain type, and only data of this type can be sent through this channel. They are directed: channels are unidirectional. If bidirectional communications are needed, two channels of opposite directions must be created between processes. They have a period which is the same as the period of the cell at the start of the channel. They are static: this means that processes connected at the other end of a channel can't change during execution, and that channels can't be created nor destroyed at runtime. Processes are also static, they can't be created nor destroyed at runtime.

In the case of complex cells, there is a special type of connection allowed, called a *shunt* which is useful when the complex cell is only a structural cell, i.e. has no body, or when some of the internal cells need to pass data to the exterior of the complex directly.

Channels do not necessarily connect only one process to another single process, they may connect many processes together. Thus three kinds of channels may be declared in a program: 1-1 channels, 1-n channels, and n-1 channels in which case an operator can be attached to the channel to reduce n values into a single one.

One of the goals of ParCeL-2 and one of its main features is to provide a topology description language in order to be able to describe easily, elegantly and concisely complex topologies. Such topology description languages have been developed, see [4, 5], but neither of them permits to construct topologies with all the features of ParCeL-2 channels.

3 The ParCeL-2 Syntax

The declaration of a type of cells consists of three main parts: an interface, a body or *behaviour*, and a *topology*.

3.1 Interface Declarations

In the interface part of a cell type, one can put type declarations, constant declarations and channel declarations. Type and constant declarations are written according to the syntax of the language chosen for the body of the cell type.

An **out** channel is always declared the same way, whether it will be connected as a 1-1, 1-n or n-1 channel:

```
OUT <data_type> <channel_name>
```

An **in** channel can be declared in three ways according to the chosen kind of utilisation, respectively 1-1 or 1-n, n-1 mailbox , and n-1 with reduction.

```
IN <data_type> <channel_name> [= <init_value>]
IN(mbox) <data_type> <channel_name> [= <init_value>]
IN(reduce, <operator>) <data_type> <channel_name> [= <init_value>]
```

The `import <cell_type>` statement permits to reuse cell code.

3.2 Body Declarations

In the body part of a cell type, anything that can be written in the chosen programming language can be written. With the exception that some keywords are reserved by ParCeL-2. That code will be executed at each superstep.

3.3 Topology Declarations

This is in that part of a cell type that internal cells are instantiated and connected according to the topology desired by the programmer. This is what differentiates elementary cells from complex cells. Complex cells embed several instances of other types of cells, which are called *internal cells*, and the topology describes how these instances are connected together and with the embedding cell. The topology description language is currently still under development. Work is underway to provide construction operators like the cartesian product of two graphs and the joining of two sub-topologies, like in Candela [6]. The language will be shown through two examples which illustrate a highly regular and a highly irregular topology. The cell types used are supposed to be declared somewhere else.

```
ring(n) of TA
{
  aux = array(1..n) of TA;
  for i in 1..n
```

```
    connect(aux[i].out1,aux[(i mod n) + 1].in1);
    connect(aux[i].out2,aux[(i mod n) + 1].in2);
    connect(aux[(i mod n) + 1].out3, aux[i].in3);
  end for;
}
ring(3) of TA;

A = new T1; B = new T2; C = new T3;
connect(A.out1,B.in1); connect(B.out1,C.in1); connect(C.out1,A.in1);
connect(A.g,g); connect(h,B.h); shunt(C.e,c); shunt(d,e.f);
```

4 Conclusion and Future Work

A programming language particularly well suited to the design of massively parallel fine grained applications has been presented. Complex computations using cellular automata, partial differential equations, finite element methods, systolic arrays or even static neural networks can benefit much from such a language, notably in portability and readability.

A first version of the runtime environment has been implemented and used to run a program implementing the game of life with a grid of 1000×1000 cells on a Fast Ethernet network of 333 MHz Sun Ultra 10 workstations using MPI as a communication library. For 50 iterations, the best sequential program took 3.3 s. The parallel version was slower, but scaled quite well, from 22.5 s to 3.89 s, using 1 to 16 processors, which is very satisfying considering that our runtime environment is not optimized at all yet. Now its behaviour must be studied with real world applications like wave propagation [7] for example.

Work on a full compiler remains to be done. It exists only in parts at the moment.

References

[1] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI-73*, 1973.

[2] G. Agha. *ACTORS, a model of concurrent computation in distributed systems.* MIT Press, 1986.

[3] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[4] Alexey Lastovetsky. mpC - a multi-paradigm programming language for massively parallel com puters. In *ACM SIGPLAN Notices*, volume 31(2), pages 13–20. February 1996.

[5] András Varga. Parametrized topologies for simulation programs. In *Western Multiconference on Simulation (WMC'98) / Communication Network s and Distributed Systems (CNDS'98)*, San Diego, CA, January 1998.

[6] Herbert Kuchen and Holger Stoltze. Candela – a topology description language. *Journal of Computers and Artificial Intelligence*, 13(6):557–676, 1994.

[7] F. Guidec, P. Calégari, and P. Kuonen. Parallel irregular software for wave propagation simulation. *Future Generation Computer Systems (FGCS), N.H. Elsevier*, 13(4-5):279–289, March 1998. ISSN 0167-739X.