The Parallel Cellular Programming Model

Paul-Jean Cagnard
Swiss Federal Institute of Technology
Computer Science Theory Laboratory
IN Ecublens, CH-1015 Lausanne, Switzerland
cagnard@di.epfl.ch

Abstract

We present a synchronous parallel programming model designed for massively parallel fine grained applications such as cellular automata, finite element methods or partial differential equations. In this model we assume that the number of parallel processes in a program is much larger than the number of processors of the machine on which it is run. We present the computational model and the communication model. We introduce the virtual cellular machine, an abstract machine implementing this programming model which requires means to simulate efficiently the execution of many processes on a single processor, and to use the available communication bandwidth efficiently. Finally, we show an example program written in a prototype language designed for programming the virtual machine.

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. Concurrent object computation traces back to the actor model proposed by Hewitt [7], later improved by Clinger and by Agha [1].

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. As a consequence, communication has to be synchronized explicitly and each transaction between two objects must be implemented as a separate communication. Mail boxes or message queues have to be managed by the underlying system. This entails that communication is not easy to implement efficiently and concurrent object systems are often restricted to coarse grain parallelism for performance reasons. As a result, concurrent object programming, while finding a wider and wider acceptance for implementing distributed systems over wide area networks [5], is still seldom used to drive massively parallel high performance computers.

Bulk-synchronous computation was first introduced by Valiant [14, 15] with the Bulk Synchronous Parallel (BSP) model. Theoretical results [15] strongly suggest that it may be implemented efficiently on many parallel computer architectures.

In the following we will present a parallel programming model that can be seen as derived from the BSP model and other synchronous object-oriented models. We will present a virtual machine which runs processes according to this model. We will also present an example program written in a prototype language for our proposed model.

2. The parallel cellular programming model

A parallel cellular program is typically composed of a large number of fine grained processes, that is processes which execute only a small number of instructions. The computation model described below is inspired from the BSP model of computation [14]. It is more precise than the BSP model because it not only allows processes to communicate, but specifies what types of communications are allowed and how processes can communicate.

2.1. Computation model

A parallel cellular program is composed, as we have seen, of a given number of processes executing concurrently. Each process execution consists, using the BSP terminology, of a sequence of supersteps. A superstep is composed of two phases: a computation phase and a communication phase. Supersteps are separated by synchronization

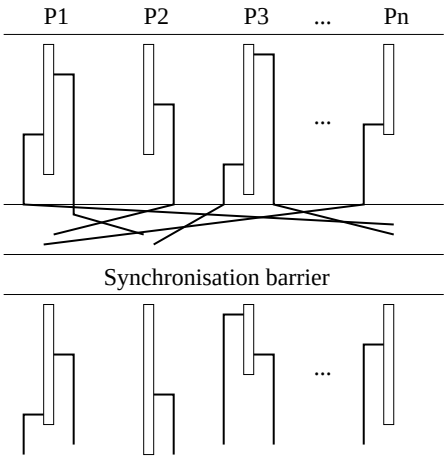

Figure 1. Program execution in the BSP model

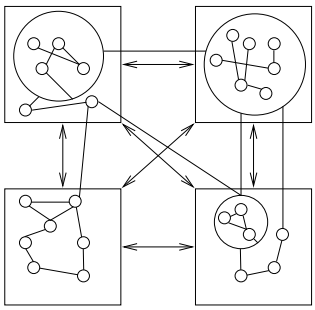

Figure 2. Program execution on four processors. Processors are represented by squares, processes by circles. Lines with arrows correspond to communication links provided by the underlying hardware or operating system, lines without arrows correspond to communication paths between processes

barriers. The execution of a program can be represented like in Fig. 1. During the computation phase of a superstep, a process executes computations which only manipulate data local to this process. These data can be local variables or data that has been received from another 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. This also implies that data sent from a given process P_1 to another process P_2 during superstep s, will only be available to the process P_2 at the beginning of the next superstep, that is superstep number $s + 1$. This type of delayed communication corresponds to the way cellular automata work. We will describe in more detail the communication model in the following section. We can observe that the parallel cellular model is an intrinsically distributed memory MIMD model.

Processes are not required to be executed in a flat structure where each process is at the same level as each other, and each process can "see" each other. 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. We can see an example of such a program in Fig. 2. This provides means for information hiding like in object oriented models. This also permits to use libraries of routines written following the same computation model, otherwise one would only be able to use either sequential external routines or parallel routines not following the superstep concept.

In a real world application, one may not want to synchronise all processes at the same frequency. In a program there is always a kind of global clock, this is the sequence of supersteps of highest frequency. A process may need

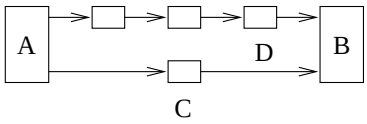

Figure 3. Reduced synchronisation frequency due to data dependencies

to be synchronised or to communicate data every n supersteps only, not at each superstep for various reasons; it could present a complex behaviour that requires n supersteps to be computed, or it could have to wait n supersteps because of data dependencies like in Fig. 3 where process C has to send data only three times slower as the processes above itself in order to feed process B at the same pace as the chain of smaller processes.

Aggregates of processes and synchronism are the main characteristic features of the parallel cellular programming model. They enable developers to design programs in a modular and hierarchical way, starting from basic simple cells to more complex ones whose behaviour is represented by a network of interconnected cells which only act on their own local information. Thus reuse of already developed cells is encouraged, and the global architecture of a program can remain clear throughout development and provide a lot of information to the runtime environment in order to optimize execution speed.

Finally, is is to be noted that the number and kind of processes remain constant during execution of a program. This implies that processes can't be created nor destroyed at runtime.

2.2. Communication model

In a parallel program processes must be able to communicate in order to accomplish a useful task. They can communicate according to several models like message passing or direct remote memory access for example. In the parallel cellular model, processes exchange data in a way similar to that of the task channel model [3]. This means that they can only communicate through channels.

A channel can be viewed as a link connecting an output port of a process which produces data to an input port of a process that consumes data. Channels have the following 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. The type of a channel can be any type allowed for a variable.
- They are directed. A channel can only carry data from an output port of a process to an input port of another process. If bidirectional communications are needed, two channels of opposite directions must be created between processes.
- They have a period which represents the minimal number of supersteps that must be executed before a new value can appear at the output of a given channel. This period corresponds to the inverse of the synchronisation frequency of the process that sends data through this channel.
- They are static. This has three consequences. First, the processes connected by a channel can't change during execution. Second, they must be declared before execution; channels can't be created at runtime. Third, channels can't be destroyed during execution.

A process can write as many different values into a given channel during a computation phase, but only the last written value will effectively be transmitted to the other end of the channel during the communication phase. Conversely, the output of a channel, which is the same as an input port of a process, always contains a valid value. This means that reading the content of a channel will never produce an error. This in turn implies that the same value will be available when reading a channel as long as no new value is sent through this channel. As data are effectively transmitted only between computation phases, the value a process gets when reading the contents of a channel doesn't change during a computation phase.

When processes are organized into aggregates, or groups, communications can only occur between processes or groups of the same grouping level or the level just above. We can distinguish two different cases involving communications with groups of processes. First, processes inside a group can communicate between them and with the group. Second, processes external to a group can communicate between them and with the group itself, not directly with any process inside the group. Thus a groups of process acts as a virtual process serving as an interface between the processes inside the group and processes outside of the group.

Three kinds of channels may connect processes:

- 1-1 channels, which connect a single process to another one;
- 1-n channels, which connect the output port of a single process to the input port of n other processes;
- n-1 channels, which connect the output ports of n different processes to the input port of a single process.

The first kind of channels implements one-to-one communications, the second kind is equivalent to multicasting because the same data is sent to the n receiving processes; this kind of channels is semantically identical to n 1-1 identical channels starting from the same processor. The third kind is a little more complex since n different values are sent to a single input port. We have thus defined two different semantics for these channels. One is equivalent to gathering, that is to say that the n differents values are combined into one using an operator associated with the channel. The other groups the n values into a set and the receiving process can extract individual values from this set, this is equivalent to a mailbox.

These different kind of channels, especially the n-1 channels, and their persistence are key characteristics of our communication model. Developers never have to worry whether a value is available in a channel or not because there is always one. Thus a program will never fail during communication operations. Persistent channels and synchronous communications provide a much simpler communication model to developers than traditional message passing, which can be considered as a low level communication model.

It is important to note that several different channels can exist between two processes. Since communications can only occur through predeclared channels, we can represent a program by a directed multigraph in which processes correspond to vertices and channels to edges. This level of abstraction enables a programmer to separate the description of the behaviour of the processes and the communication or interconnection topology. The use of graphical tools to design programs can thus be envisioned. Novis [4] is a tool that permits to design parallel programs graphically, but its goal was more oriented towards visual simulation than towards high performance.

Finally, we can observe that a process in the parallel cellular model can be viewed as an object in the sense of object-

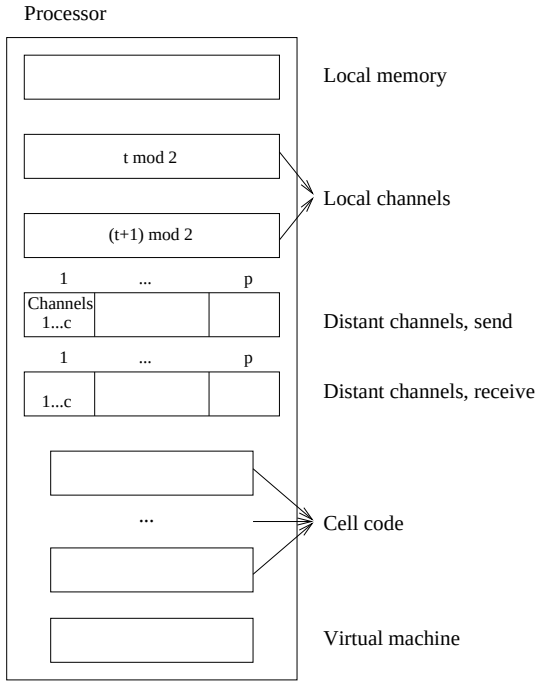

Figure 4. Architecture of the virtual cellular machine

oriented languages, with private data and code, and public data that are represented by channels.

3. The virtual cellular machine

The virtual cellular machine is the execution environment in which a parallel cellular program is run. Since we have assumed massively parallel fine grained programs, there will be much more processes than processors, so the virtual machine must find an efficient way to simulate execution of these processes on the available processors, and to use the available interprocessor communication bandwidth efficiently since fine grained programs tend to communicate a lot more than they compute. The general architecture of the virtual machine on one processor is represented in Fig. 4. The runtime provided by this virtual machine is fundamentally different from that of hardware cellular machines like, for example, the Connection Machine [9]. While the Connection Machine is a SIMD machine, permitting only the execution of identical programs on all processors, where all these processors act on a tiny part of a global data structure, our virtual cellular machine offers a kind of MIMD environment, where different processes can be executed on different processors, and where a single processor can run many processes in simulated concurrency. The most efficient way of simulating the concurrent execution of multiple processes on a single processor, especially since all processes must have finished their computation phases for the program to proceed, is looping through them. Thus on a processor one can find the instructions for all processes attributed to this processor. In addition there is some code for the management of the virtual machine, mainly to effectively send the data between supersteps. The virtual machine will thus execute in turn the code corresponding to every process that is to be run on a given processor, transmit data to other processors, and start again until the end of the program.

In order to use the interprocessor communication bandwidth efficiently, the data that are to be sent to another processor are grouped into contiguous locations in memory. Thus a single large message can be sent between two processors at the end of a superstep instead of many small ones. We can see in the figure that memory is divided into several specific zones. First there is a zone for local data which are manipulated by processes but never sent between processes. Then there are two zones used for data that corresponds to contents of channels connecting processes running on the same processor. In that case no interprocessor communication is needed and the data are only copied from one zone into the other at the end of a superstep. Of these two zones, one is the current data, available during computation phases, and the other is the future data, which will be available during the next computation phase. They play the roles of current and future data in alternance, in order to avoid copying whole zones of memory at each superstep. Then there is the zone of memory containing data to be transmitted to other processors. It is subdivided into smaller zones corresponding to a specific processor. In each of these smaller zones we have the data corresponding to individual channels. Finally there is the zone containing data received from other processors. We could have put the local communication memory zones into these last two, but for the sake of simplicity we have kept them as individual zones.

The virtual machine doesn't do any load balancing. It doesn't need to since processes can't be created or destroyed and the communication topology doesn't change at run time. This is the role of the compiler to place the processes on the processors, taking into account characteristics such as the computing load and the communication load of individual processes. The main idea is to let the programmer write a program according to the mathematical or physical model he uses. He doesn't need to artificially group smaller processes into coarser grained ones as he generally has to do and especially in ParCeL-1[16], this is the role of the compiler and the virtual machine, with the aid, if he so desires, of the programmer to force the placement of specific processes on certain processors.

4. An example program

We will present here a typical example of a massively parallel fine grained program: the simulation of moving particles on a discrete grid. We assume the reader is familiar with cellular automata. The simulation is elementary but sufficient to show the features of our computing model. Each particle has a position in the discrete grid and a speed that can be oriented in four directions: north, south, east ,and west. Each cell of the grid can contain only one particle at a given point in time, has four neighbours, from which particles can come and to which particles can go, handles the collisions between particles. Thus a cell can be empty or contain a particle with its own speed. Basic grid cells will be grouped into stripes in order to show aggregates of cells.

In this program, each cell corresponds to a process and each cell is connected to its neighbours by channels in both directions. As said before, the description of the processes can be entirely decoupled from the description of their interconnection topology. The behaviour of processes is described using an extension of the C language, and the topology is described in a graph file format called GML [10]. The language is under development and is called ParCeL-2. We could have extended any other language since we only need a few constructs to express parallelism and communication channels. Thus we benefit of all the basic work needed for the creation of a programming language.

Many languages exist already which can be used to program cellular automata or artificial neural networks, see [13, 2, 11] for example. But either they are too tied to their specific domain or their goal was expressiveness and dynamism rather than performance. In particular, cellular languages only allow to program regular grids of processes, not any desired topology; and neural network languages try to offer too much dynamism in the topology of processes, which leads to poor performance.

Here is the partial code of a typical basic cell from our simulation. Repetitive and unimportant parts have been removed. In particular, we assume that a function named `handle_collision` exists and computes the result of the collision of several particles at the place of a cell. The different `speed` variables encode the speed of particles moving away from the grid cell after a collision or of the particle present in the cell when no collision occurs. The `direction` variable stores the direction of movement of the particle in a cell.

```
celltype grid_cell
{
  IN  int N,S,E,O;
  OUT int N,S,E,O;
  int speed_N, speed_S, speed_E, speed_O;
  int speed;
  int direction;

  switch (handle_collision())
          {
          case 0 :
            write(N, speed_N);
            write(S, speed_S);
            write(E, speed_E);
            write(O, speed_O);
            break;
          case 1 :
            switch (direction)
              {
                case 0 :
                  write(N, speed);
                  break;
                ...
                case 3 :
                  write(O, speed);
                  break;
               }
          }
}
```

A stripe cell is composed of a rectangular part of the whole grid. In this case a stripe cell is only a structuring element, which means that it doesn't act on the data that pass through it; it hides the complexity of its behaviour. A compiler can optimize the execution speed of the whole program by flattening its structure. We assume that the whole grid is a torus. The various `inside` channels are the ones that come from and go to the basic grid cells hidden inside a stripe cell, while `outside` channels are the ones which enable communication with external cells.

```
celltype stripe
{
  int width = 1000;
  int height = 100;
  int i;

  IN  int inside_in_N[width],
          inside_in_S[width],
          inside_in_E[height],
          inside_in_O[height];
  IN  int outside_in_N[width],
          outside_in_S[width],
          outside_in_E[height],
          outside_in_O[height];
  OUT int inside_out_N[width],
          inside_out_S[width],
          inside_out_E[height],
          inside_out_O[height];
  OUT int outside_out_N[width],
          outside_out_S[width],
```

```
          outside_out_E[height],
          outside_out_O[height];

  for (i=0; i<width; i++)
    {
      write(outside_out_N[i],
        read(inside_in_N[i]));
      write(outside_out_S[i],
        read(inside_in_S[i]));
    }
  for (i=0; i<height; i++)
    {
      write(outside_out_E[i],
        read(inside_in_E[i]));
      write(outside_out_O[i],
        read(inside_in_O[i]));
    }
}
```

Here is the interconnexion topology. Again we have suppressed repetitive and unimportant parts. We can see the declaration of two nodes corresponding to two grid cells of the particle simulation, and the declaration of one channel from node number 1 to node number 2. We observe that channels are declared in the nodes as well as in the edges. This is due to the fact that input or output ports have to be present in the processes, or nodes, in order to be connected by channels, or edges. The hierarchical structure of the program appears in the `graph` declaration inside the first node. Connections between internal cells of type `grid_cell` can be seen in the declaration of the `edge` whose `id` is 1.1, while connections between cells of type `grid_cell` and cells of type `stripe` can be seen in the declaration of the `edge` whose `id` is 1.2.

```
graph [
  node [
    id 1
    type stripe
    inchannel inside_in_N int 1000
    ...
    inchannel inside_in_E int 100
    ...
    outchannel outside_in_N int 1000
    ...
    graph [
      node [
        id 1.1
        type grid_cell
        inchannel N int
        ...
        outchannel N int
        ...
      ]
      node [
        id 1.2
        ...
      ]
      edge [
        id 1.1
        source 1.1
        outchannel S
        target 1.2
        inchannel N
      ]
      ...
      edge [
        id 1.2
        source 1.1
        outchannel N
        target 1
        inchannel inside_in_S 0
      ]
      ...
    ]
  ]
  node [
    id 2
    ...
  ]
  ...
  edge [
    id 1
    source 1
    outchannel outside_out_N 0
    target 2
    inchannel outside_in_S 0
  ]
  edge [
    id 2
    ...
  ]
  ...
]
```

We ran a program with a grid of 1000×1000 cells on a network of 333 MHz Sun Ultra 10 workstations using MPI as a communication library. The speed-up we were able to achieve was nearly linear up to 16 processors, which is very satisfying considering that our virtual cellular machine is not much optimized yet.

5. Conclusion and future work

We have presented a programming model particularly well suited to the design of massively parallel fine grained applications. Cellular automata, partial differential equa-

tions and finite element methods are widely used for complex calculations or physical simulations, and we believe they can benefit from such a model with its associated language. Even programs using static neural networks may benefit from this programming model. Since programs are static in the sense that neither processes nor channels can be created, modified or destroyed at runtime, it is easy to represent a parallel program by a multigraph. Then we can use any graph algorithm like graph partitioning techniques to place processes on the available processors, unless the programmer decides he wants to place the processes himself.

We have implemented a first version of the virtual cellular machine that works for simple programs like the game of life or moving particles simulations. The first results are promising. We now have to study its behaviour with less regular programs, that is programs containing processes with varying computation and communication loads. This includes tests with real world applications like wave propagation[6] for example. Work is underway to use the virtual machine for linear algebra computations like matrix multiplication with systolic algorithms as described in [12].

It would be interesting to use the BSP library from Oxford university [8] instead of MPI as a communication layer.

Finally, work on a full compiler remains to be done. It should start from the descriptions of the processes and of their interconnections and build a graph representing the program. From there it can proceed to optimise placement of processes, and generate C code that can be compiled and linked with the virtual machine.

6. Acknowledgements

We would like to thank the Swiss National Science Foundation for supporting this work under grant 21-49519.96.

References

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

[2] J. D. Eckart. Cellang 2.0: Language Reference Manual. *ACM SIGPLAN Notices*, 27(8), Aug. 1992.

[3] I. Foster. *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering*. Addison Wesley, 1994.

[4] E. P. Glinert and C. D. Norton. Novis: A visual laboratory for exploring the design of processor arrays. *Journal of Visual Languages and Computing*, 3(2):135–159, June 1992.

[5] R. Guerraoui. Les langages concurrents à objets. *Techniques et Sciences Informatiques*, 14(8):945–971, 1995.

[6] 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, Mar. 1998. ISSN 0167-739X.

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

[8] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. Technical Report May, Oxford University Computing Laboratory, 1997.

[9] W. D. Hillis. *The Connection Machine*. The MIT Press, 1985.

[10] M. Himsolt. *GML: Graph Modelling Language*. Universität Passau, 94030 Passau, Germany, Dec. 1996. Draft Version.

[11] H. Hopp and L. Prechelt. CuPit-2: A portable parallel programming language for artificial neural networks. In *1997 IMACS Conference on Applications of Computer Algebra*, 1997.

[12] W. F. McColl. Special purpose parallel computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 261–336. Cambridge University Press, 1993.

[13] G. Spezzano and D. Talia. CARPET: A programming language for parallel cellular processing. In *Proceedings 2nd Europ. School on PPE for HPC*, pages 71–74, Apr. 1996.

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

[15] L. G. Valiant. Genaral purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18, pages 945–971. Elsevier Science, 1990.

[16] S. Vialle, T. Cornu, and Y. Lallement. ParCeL-1: a parallel programming language based on autonomous and synchronous actors. *ACM Sigplan Notice*, 31(8):43–51, Aug. 1996.