Design and implementation of a parallel cellular language for MIMD architectures

Stéphane Vialle
Supélec
2 rue Edouard Belin
57070 Metz
France
vialle@ese-metz.fr

Yannick Lallement
Carnegie Mellon University
Human-Computer Interaction Institute
5000 Forbes Avenue
Pittsburgh PA 15213, USA
yannick@cmu.edu

Thierry Cornu
Simulog
60 Rue Lavoisier
38330 Montbonnot St Martin
France
tcornu@grenoble.simulog.fr

Abstract

In this paper we present a new language, called ParCeL-1, intended to make easier the implementation of computation-intensive and artificial intelligence applications on highly parallel MIMD architectures. The computational model of ParCeL-1 is object-oriented and synchronous, based on virtual processors called cells that compute and communicate alternately. The current prototype is implemented on several multi-processor systems (Cray T3D, Intel Paragon, Telmat T-Node, workstation networks using PVM, and SGI Power Challenge Array). Several applications written in ParCeL-1 are described together with their performances obtained on a 256-processor Cray T3D.

parallel language, actors, neural networks, parallel implementation

1 Introduction

Parallel programming is an attractive way to speed up heavy applications, and run them today in acceptable time instead of having to wait for a technology upgrade. Unfortunately, writing parallel programs is not a trivial task; different methods exist, like data parallelism, control parallelism and original parallel models like actors. Each method has an associated set of programming languages.

Data parallelism consists of distributing data among a lot of elementary processors, each of them executing the same operations on its piece of data. Some compilers provide automatic or semi-automatic distribution of data, as well as parallelization of loop constructs. Tedious tasks like the distribution of code and data structures are, to some extent, automated. Unfortunately, these compilers currently handle only loop nests. Moreover, they work efficiently mainly with regular data structures without indirect addressing. Data parallel programming models are, therefore, suitable mainly for scientific computation and regular problems, like dense matrix computation. They were implemented first on SIMD (Single Instruction Multiple Data) architectures, but are today implemented on MIMD (Multiple Instruction Multiple Data) architectures too.

Conversely, control parallel models allow parallelization of irregular problems. Control parallel models can be implemented only on MIMD architectures. They consist of decomposing a problem into a set of concurrent and communicating tasks that will cooperate to solve the problem in parallel. But a classical control parallel language still remains difficult to program. Both for message passing and shared memory languages, the decomposition of code and data into concurrent tasks, the synchronization operations and the communication between tasks are frequently left to the programmer. A long term goal for parallel computing is automatic parallelization, where all those difficulties are taken care of by the compiler.

By incorporating parallelism in their computing models, some concurrent object or actor languages are a step in that direction. The ParCeL-1 project began in 1989 and was oriented in that way. At that time we worked on Transputer-based multiprocessor machines, with very limited software environments, in order to implement various artificial neural networks. We decided to develop a tool which would allow us to:

- parallelize and speed up parallel artificial intelligence (AI) applications like artificial neural networks and semantic networks,
- write parallel programs in an easier way than on our Transputer native environments,
- write portable programs which hide architectural differences.

We defined a new parallel computational model, and we implemented the corresponding parallel virtual machine (VCM-1). Finally, we defined and implemented a high level parallel language (ParCeL-1) in order to easily program our virtual machine and to test our computational model.

In 1993, during the realization of the project, we slightly changed our initial objectives: we got interested not only in AI applications but also in scientific computation. We ran some benchmarks on these two kinds of applications in order to evaluate and optimize our parallel language. The main features of our language that we tried to evaluate in 1995 and 1996 were the speed up obtained on several applications, the possibility to quickly write some parallel programs, and the possibility to run these programs on different parallel machines.

In 1997 we enumerated some limitations encountered, and we decided to define and implement a new version of the language. This paper presents several aspects of this project; section 2 introduces the computational model associated to ParCeL-1; section 3 describes ParCeL-1's syntax and gives an example of a program; section 4 situates ParCeL-1 among related programming models and languages; section 5 describes the implementation of ParCeL-1 on parallel computers; section 6 gives the benchmark criteria that are used in section 7 on numerical applications and section 8 on AI-related applications.

2 ParCeL-1 computational model

We named our computational model and our virtual machine the Virtual Cellular Machine (VCM-1) [9]. In this section we describe the main features of the VCM-1: basic principles, dynamic cell graph, cyclic running, and communication protocol.

2.1 Basic principles and components

The basic component of the VCM-1 is the cell. Each cell is an instantiation of a cellular type, like objects and object types in object-oriented languages. The cellular types of a VCM-1 program are static and declared in a ParCeL-1 program, and the cells are dynamically created at run-time.

A cell contains some parameters defined at its creation, some local variables, some communication channels and some execution rules composed of a boolean condition and of an action part (`condition ==> action`), as shown in figure 1. All of these components are declared by the programmer in the cellular types. The exact syntax of a ParCeL-1 cellular type declaration is given in section 3.1.

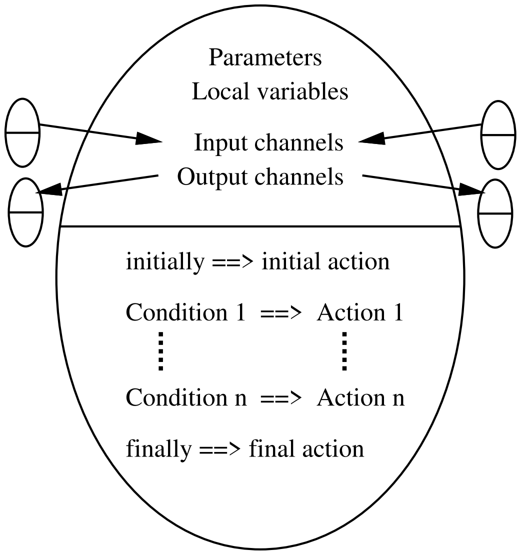


Figure 1: General constitution of a cell

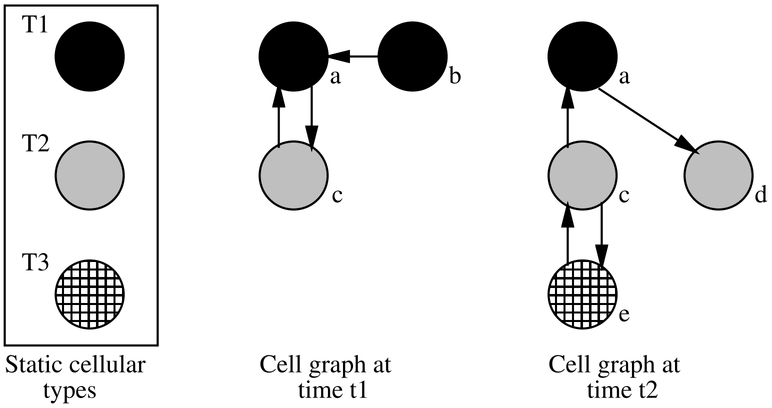


Figure 2: Example of cellular program: with static definition and dynamic execution

2.2 Dynamic cell graph

At the beginning of the execution only one cell is created (of the special `main` type). This cell can create others, which can themselves create other new cells. Dynamic cell graph evolutions are programmed by specific instructions named *requests*, whose implementation is explained in detail in section 5.4. Three requests are sufficient to create any cell network topology: creation of a new cell, destruction of a cell, and connection between two cells. Some requests need a cell *registration* as parameter, in order to identify the target cells in the cellular network. Figure 2 shows an example of a cellular program with three cellular types and two steps of execution with different cell graph states. In addition to these cell graph manipulation features, a cell can be set to sleep or to wake dynamically by other cells, and a channel can be disconnected from another using advanced requests.

The cells communicate only by sending messages on their output channels and receiving others on their input channels. The channels of a cell are defined in its cellular type. Therefore, two cells of the same type will share the same channel structure, but the channel contents are proper to each cell. Two cells that want to communicate have to explicitly connect their channels prior to exchanging messages. The communication graph is a dynamic and unidirectional graph between cells which evolves during the execution.

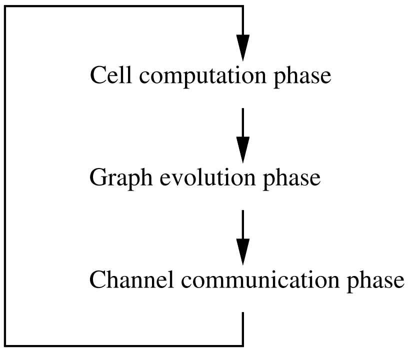


Figure 3: VCM-1 cycle repeated by any program

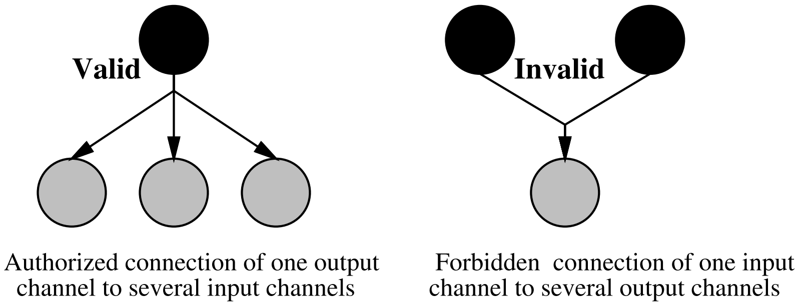


Figure 4: Example of authorized and forbidden connections

2.3 Cyclic running of the VCM-1

A VCM-1 program runs by cyclicly executing three phases (figure 3): cell computation, cell graph evolution and channel communications. It repeats these phases until executing a `halt` instruction. During the cell computation phase, some graph evolution requests are generated (cell creation, destruction, ...), some values are written in output channels and some input channels are read. The evolution requests will be processed only during the evolution phase, and the values written in the output channels will be propagated to connected input channels only during the communication phase.

This mechanism allows separation of computation and communications which was an important feature in our first Transputer multiprocessor implementation. Moreover, the cell graph state remains unchanged during a cell computation phase: the connection graph and the values of the input channels do not change during these phases. This is a main feature of our computational model, allowing easy implementation of programs like artificial neural networks or regular scientific computation; in these applications, a lot of new values are computed from old values, before becoming old values at the next iteration.

In a computation phase each cell can execute in parallel with all the others. The order has no importance as a cell execution acts on the cell networks or on input channels of other cells only after the end of the computation phase. This parallel capability has been exploited to implement a VCM-1 on parallel computers in order to parallelize cell computation.

2.4 Communication protocol

The input and output channels of a cellular type are defined at compilation time, but there is no limit to the number of channels that one cellular type can contain. An output channel can be connected to several input channels (without limitations), but an input

```
typecell filter (int init) {/* cell type definition with one param */

int local;                 /* one integer local variable         */

in  int input;             /* one integer input channel          */
out int output;            /* one integer output channel         */

initially ==> result = 0;                 /* initially rule     */

x == 1 ==> result = 100;                  /* standard  rule     */

x == 2           ==> tmp = InputChannelA;   /* standard sequence  */
TRUE             -(1)-> tmp = tmp * InputChannelB;
InputChannelC > 0 -(4)-> result = tmp * InputChannelC;

finally ==> OutputChannel = result;         /* finally rule       */
}
```

Figure 5: Example of a cellular type with 4 standard rules, one of them including two associated rules that form a sequence

channel must be connected to only one output channel, as illustrated in figure 4. This way, an output channel can broadcast its value to a set of input channels.

Each channel can store only one piece of data per computation phase. If several values are written in an output channel during the same computation phase, then only the last one remains and is routed to connected input channels at the end of the cycle. The value is not erased from the output channel when it is routed to input channels. It remains and is routed at each cycle until it is overwritten or the channel is destroyed.

This communication protocol, with a dynamic and explicit communication graph, and with no message queues associated with channels, is well adapted to programs making cycles of computation-communication after an initialization step, like artificial neural networks.

3 Syntax and example of application

In this section, we present in more detail the syntax of a cell declaration and we explain the execution of a ParCeL-1 program with a simple example: the prime number generation program listed in figure 6.

3.1 Syntax of a cell declaration

Figure 5 gives an example of an actual cell declaration. The syntax is similar to the syntax of a function definition in C [28]. First comes the name of the cell type and its parameters, then the local variables, the communication channels (`in` for the input channels and `out` for the output channels) and the execution rules. There are two types of execution rules: standard and associated.

Standard execution rules are composed of a condition and an action part with the syntax: `condition ==> action`. The condition part can be one of the `initially` and `finally` keywords or can be a boolean expression. A cell can have one initially and/or one finally rule. `initially` is true only at the first activation of the cell, and `finally` is true only at its last activation (when the cell is killed, by itself or by another one). Figure 5 has one initially and one finally rule. These rules are useful for initializations and clean-up operations. The other rules (with a boolean expression in their condition part) are used for all other computation phases. Figure 5 has two such rules, with conditions `x == 1` and `x == 2`. Each of these rules may have a priority and when the cell determines which rule it has to execute, it selects the highest priority rule whose condition is true. If several rules match that criterion, the first one (in the order of the program) is selected. The two standard rules in figure 5 have no explicit priority, so the first one has the highest priority. When it is selected, a standard rule is executed in a single computation phase.

Associated rules, with the syntax: `cond -(n)-> action` allow the user to easily program actions planned through several cycles. In figure 5, the standard rule whose condition is `x == 2` has two associated rules. The set of a standard rule with some associated rules is called a sequence. When the selected standard rule has some associated rules, then at the next cycle the cell does not check its standard rules, but checks only the following associated rule of the sequence. The `-(n)->` operator means that the cell has to wait for `n` cycles before executing this associated rule (it will spend `n-1` cycles doing nothing). When checked, if the associated rule condition is true the action part is executed. Otherwise, the sequence is interrupted. When the last associated rule has been executed the sequence is finished. When the sequence is finished or interrupted, the cell again checks its standard rules at the following cycle.

3.2 Example of ParCeL-1 program

The basic principle of the prime number program listed in figure 6 is to use a series of filters, each filter being initialized with a prime number and filtering each number divisible by that prime [6]. The filters are arranged in a pipe-line (see figure 7) and run concurrently.

This program is composed of two cell type declarations: `main` and `filter`. The `main` cell is a generator of integers that are output on its `new_int` channel. When the program is started, one instance of the `main` cell type is created.

The `main` cell starts by creating a first `filter` cell in its `initially` rule (line: `f = create filter(2)`), which will filter any integer divisible by two. In order to manipulate this new cell, its registration is stored in a registration variable (`f`). It is used at the next instruction in order to connect the new cell (the filter) to the main cell (the generator). This first filter cell will let only odd numbers pass. The first non filtered number is three. Thus, three is a prime number, and a new `filter` cell is created by this first `filter` cell that will filter all the numbers divisible by three. The process is iterative: each time a new `filter` cell is created it prints the number it will filter, which is prime, and filters all the multiples of that prime number.

The communication pattern between the cells is shown in figure 7. The output channel `new_int` of the `main` cell is connected to the `input` channel of the first `filter` cell. Then, the `output` channel of each `filter` cell is connected to the `input` channel of the next `filter` cell.

4 Related languages

This section situates ParCeL-1 among several kinds of programming languages it is closely related to: concurrent object-oriented languages, actor languages, bulk synchronous model and connectionist languages.

4.1 Concurrent object-oriented languages

Object-oriented programming was initially developed without any relationship to parallel programming models. The main distinct feature of object-oriented languages is inheritance and not concurrency. However, object programming introduces a natural partitioning of

```
#include <hcifunc.h>
#define MAX_INT 1000

typecell filter (int init) { /* cell type definition with one param */
 in  int input;            /* one integer input channel            */
 out int output;           /* one integer output channel           */

 registration filter next; /* one registration on a filter cell     */

 initially ==> {           /* rule executed on creation of the cell */
   printi(init);           /* printing of the init parameter       */
   printnl();              /* new line                              */
   next = NULL;            /* no next cell for now                  */
 }

 TRUE ==> {                /* new rule, always executed             */
   if (input % init != 0)  /* if the input is not divisible by init */
     if (next != NULL)     /* and there is a next filter cell...    */
       output = input;     /* then input is transmitted to this cell*/
     else {                /* if there is no next filter cell...    */
       next = create filter(input); /* we create a new one which    */
       connect input  of next       /* we connect to ourselves      */
            to output of self;
     }
   else output = 0;
 }
}

typecell main () {          /* main cell type definition            */
 out int new_int;           /* one integer output channel, new_int  */

 registration filter f;     /* a registration on a filter cell      */

 initially ==> {            /* rule executed on creation of the cell*/
   f = create filter(2);    /* creation of a filter cell...         */
   connect input   of f     /* and connection to this main cell     */
        to new_int of self;
   new_int = 2;
 }

 new_int < MAX_INT ==>
   new_int += 1;            /* increase new_int at each cycle       */

 new_int = MAX_INT ==>
   halt;                    /* stop when MAX_INT is reached         */

}
```

Figure 6: ParCeL-1 prime number program.

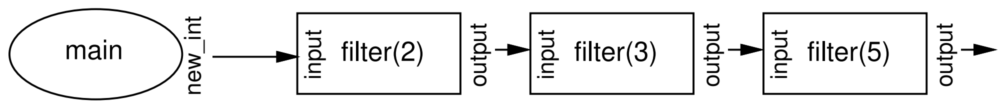


Figure 7: Communication graph of the prime number program.

data and tasks, so that the concept of an object provides a good metaphor for parallel computing models. Therefore, parallel extensions have been proposed for most object-oriented languages (for instance C++ [8], Mentat [11], Smalltalk [21]). In concurrent object-oriented languages, the concept of inheritance as such is completely independent from the semantics of parallelism.

Nearly all models based on concurrent objects use asynchronous execution models, i.e., models where computation and message sending are not synchronized. Consequently, 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. As a result, communication is not easy to implement efficiently and concurrent object systems are often restricted to coarse grain parallelism for performance reasons. Then, concurrent object programming, while finding a wider and wider acceptance for implementing distributed systems over wide area networks [5], is still seldom used in massively parallel high performance computers.

ParCeL-1 was designed first for transputer based systems with poor programming environments. We decided not to provide any inheritance mechanism in ParCeL-1 in order to make its implementation easier. However, since objects in ParCeL-1 are statically typed, it would be feasible to extend it with inheritance functionalities similar to those of compiled object languages like C++.

4.2 Actor languages

Actor-based programming traces back to the computational model proposed by Hewitt [12] and later improved by Clinger and Agha [1]. Actors are conceived as *active* objects, i.e., objects with their own resources and activity. Actor languages may or may not provide inheritance. When available, inheritance does not directly influence the way concurrency is handled. Most actor languages provide not only inter-object parallelism, i.e., concurrency between different objects, but also often intra-object parallelism, i.e., objects able to process several requests simultaneously [1]. In the ACT language [25], every single computation (for instance the addition of two numbers) implies transactions between several actors.

The communication and synchronization protocols of actor languages may prove tricky to implement efficiently on multiprocessor machines; as a matter of fact, actual multiprocessor implementations of actor languages appeared only after 1990. The HAL language based on the CHARM virtual machine [14] is a system in the lineage of the work of Hewitt, Clinger and Agha. The successive variants of the ABCL language [30] were developed by the team of A. Yonezawa. In ABCL several design choices are more conservative in an attempt to facilitate an efficient parallel implementation. ABCL/1 is a parallel extension of Common-Lisp. Simple data structures and operations are implemented in Lisp without involving any actor-based mechanism. Similarly, an actor can process only one request at a time; however, this processing may be interrupted to switch to another task. Several versions of ABCL actually have a multiprocessor implementation.

ParCeL-1 is similar to actor-based languages but with a synchronous computational model. Several multiprocessor implementations of ParCeL-1 are fully operational. The underlying VCM-1 virtual machine has been available since 1992 and the first implementation of ParCeL-1 itself since 1994.

4.3 Bulk-synchronous computation

Bulk-synchronous computation was first introduced by Valiant [26, 27] with the Bulk synchronous programming (BSP) model. Computation in a BSP program consists of a sequence of super-steps; each super-step is a sequence of steps followed by a synchronization barrier at which non-local communications take effect. Theoretical results [27] strongly suggest that it may be implemented efficiently on many parallel computer architectures, and actual programming tools using BSP were developed [10, 7].

Meanwhile, new MIMD architectures appeared, such as the Cray T3D and T3E, in which virtual shared memory as well as synchronization barriers are hardware-implemented. The most efficient way to program these machines is to use asynchronous remote memory access, also called one-sided message passing, and synchronization barriers. This programming style closely matches the BSP model, and implementing a BSP library on top of these machines is extremely simple [20].

Some features of the BSP model are common to the ParCeL-1 model, in particular the cycles / super steps of the program executions and the non-local communications treated only at the end of the cycles / super steps.

4.4 Connectionist languages

One of our original goals in designing ParCeL-1 was to make the programming of neural networks easier. Three kinds of tools can be used for this task: simulators, dedicated languages and general purpose languages. In this section, we compare ParCeL-1 viewed as a connectionist language to other such languages.

A language can be more or less versatile according to the variety of networks that can be built, and the variety of agents that can compose the network. Some languages, such as Aspirin [19] or NeuDL [23], implement only one kind of model (usually back-propagation networks). Other languages, such as NIL [4], propose only one type of elementary agent (the neuron). ParCeL-1 is completely unconstrained, that is, any kind of neural network can be implemented using any kind of basic agents (e.g.: synapse, neuron, group of neurons, layer, etc.).

Synchronization mechanisms also differ from one language to another. In CuPit [22] or Condela-3 [2], a *central agent* applies a given function to a given set of agents (e.g. neurons) at each cycle. NIL's agents [4] follow a CSP-like [13] rendez-vous mechanism and are activated when they receive data. ParCeL-1's agents are activated at each cycle, thus, when necessary, synchronization must be implemented explicitly. For example, it is possible to create a manager cell connected to and controlling the other cells.

Finally, parallelism is another key issue: connectionist applications are very demanding for computing power, but parallel languages are still rare. CuPit [22] implements an SIMD parallelism on MasPar architectures by triggering several identical agents at the same time. ParCeL-1 is implemented on MIMD architectures, and different agents can perform different computations at the same time.

5 Implementation

This section deals with the implementation of ParCeL-1 on parallel computers. It will describe the multi-process management, the load balancing method, the implementation of the communication channels, the implementation of the cell graph evolution requests, and, finally, the code generation.

5.1 Multi-process management

A ParCeL-1 cell may have many variables and many instructions per rule, but more often it has only a few of each. Because of this usually small size, a typical cell cannot be imple-

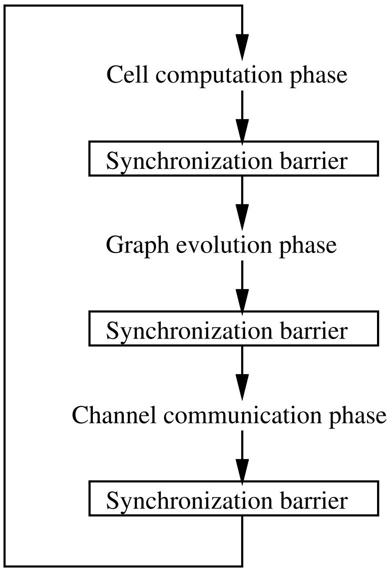


Figure 8: ParCeL-1 cycle for implementation on parallel computer

995	995	994	995	995	996	997	998
998	998	998	999	998	997	995	995

Table 1: Number of cells on 16 processors running a 15-queens program

mented efficiently as an independent process; the process management overhead would be too important compared to the cell execution time. We decided to implement ParCeL-1 on parallel computers using only one process per processor. Each process manages a subset of cells and repeats the three phases of each cycle.

According to the computing model, each cell executes its computation phase without any synchronization with the others; therefore, the different processes are not synchronized at each instruction during the cell computation phase. However, they need to get through each phase together, so we added three explicit synchronization barriers to the theoretical ParCeL-1 cycle, as shown on figure 8, in order to synchronize the processes at the beginning of each phase.

Since the cellular network remains unchanged during a cell computation phase (see section 2.3), cells can be activated in any order, and we can activate several cells in parallel from several processors. Each processor activates its cells sequentially but in parallel with the other processors. The cell computation phase ends only when it is finished on every processor.

Finally, a cell is encoded simply as a registration number, a private data structure, and pointers to a subroutine shared by all cells of the same cellular type, not as an entire process. It is a very light structure which is quickly managed.

5.2 Load balancing

The first parallel computers we used had distributed memories, and we had to choose between encouraging load balancing with a good distribution of cells among processors in order to reduce processor waiting time, or grouping together the connected cells that needed to communicate in order to reduce communication time. These two strategies (distribution/grouping) are opposite, and we chose to favor the load balancing which seemed easier to realize, as it is difficult to detect to which cell another cell is going to connect dynamically and to group them in advance on the same processor.

Since the cell graph is dynamic and can depend on data and computing results, the load balancing cannot be determined at compilation time. We need a dynamic load balancing mechanism; this can be achieved with or without the possibility of cell migration. The use of a cell migration mechanism seems to be more powerful, but migration (cell-, object-, process-, ...) is a costly mechanism. It is difficult to know if the time spent moving a cell will be smaller than the time saved by better load balancing. We preferred to use a placement algorithm that can install a new cell on one of the least loaded processors in very little time.

In order to implement that, each processor evaluates its load at each cycle from its number of cells. We assume that the different cell types are present on each processor and are uniformly distributed in the processor network. In this case, the number of cells on a processor is representative of its load. At the end of each ParCeL-1 cycle, each processor broadcasts its load evaluation to every other; after this operation, every processor knows the load repartition in the network. The possible load imbalance between processors is used by each processor to determine the host processor of the newly created cells. In order not to slow down the ParCeL-1 cycle with control data communication, no more load balancing information is exchanged until the end of the next cycle. So, when a processor executes a request for creation of a new cell, the choice of the reception processor is done using only the load repartition at the previous cycle.

We tested this placement algorithm on an N-Queen program (see section 7.3). That program does a parallel tree-search by splitting the original tree into subtrees that are searched in parallel (see section 7.3). On this example, the splitting operation is done recursively creating specialized splitting cells. On a 15-Queens problem, splitting the initial tree until depth 3 creates approximately $16,000$ cells. On 16 processors of a T3D parallel computer, the experimental cell number difference between processors is less than 0.5%, as shown on table 1.

5.3 Channel communication implementation

During a cell computation phase, a cell can store data in its output channels; this data will remain private to the cell until the next communication phase. During the communication phase, the output channels become readable by every cell that has a connected input channel. During the next cell computation phase, the input channels will read the output channel values of the previous cycle.

This channel communication model can be implemented using two kinds of data structures: cell private buffers to save the current output channel values, and a ParCeL-1 shared memory readable by every cell to save the previous output channel values. The private data structures are dumped into the shared one at communication phase. Figure 9 illustrates this mechanism.

On shared memory multiprocessors these data structures are straightforwardly implemented, and the dumping operation is just a memory copy. On distributed memory multicomputers this mechanism needs more support software. The private buffers of all the cells of each processor are grouped into an output table in the local memory of the processor, and each output table is broadcast to every other processor. This way, any previous output channel is duplicated on every processor and can be read by any input channel.

In order to minimize the size of the output tables, we implemented a garbage collection mechanism that maintains the output channels in a continuous space from the top of the

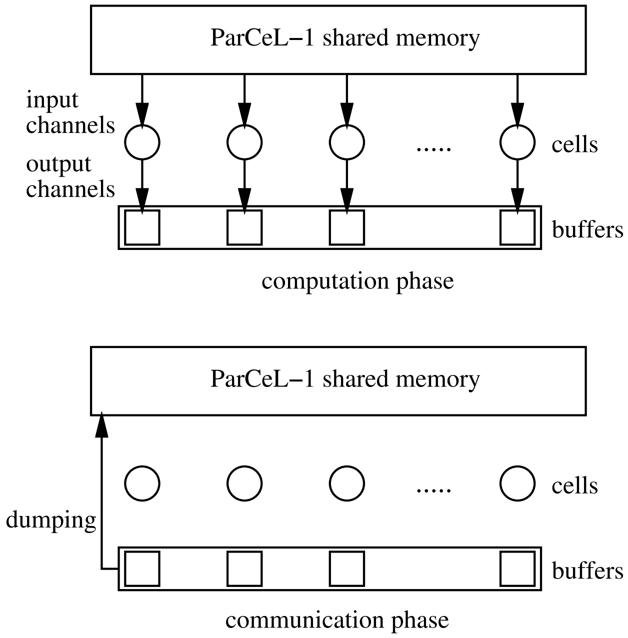


Figure 9: Basic data structures for ParCeL-1 channel communication protocol

table. The tables are sent with a single broadcast instruction, and only the minimal data are transmitted, thus minimizing transmission time.

5.4 Implementation of cell graph evolution requests

Cell graph evolution requests emitted during cell computation phases are stored in order to be executed later, like channel communications. A request is emitted by a cell and concerns at least one other (e.g. set another cell to sleep). The two cells can be hosted by different processors. It is therefore necessary to route the requests through the processor network. Unlike in channel communication, a request is not broadcast, it is addressed by one processor to another.

Most requests need several route and process operations, because two processors are sequentially engaged in their execution and need to perform consistency checks. For example, a connect request between an input channel of cell A and an output channel of cell B needs the intervention of the host processor of cell A and the host processor of cell B, and needs to check if cells A and B are still alive before attempting to connect them. The connection request is actually one of the longest to execute and takes three route and process operations.

Evolution requests are executed during the second phase of each VCM-1 cycle, which is composed of three route and process sub-phases. The execution process always checks if the cell network can support a request before executing it.

5.5 ParCeL-1 code generation

We chose to write a compiler that generates C code instead of assembler code, and to call a C compiler. This makes ParCeL-1 compilation slower than if it was a direct compilation and ParCeL-1 binary executable not as well optimized. But the use of C as an intermediate language has three main advantages:

- C is very common, so our implementation of ParCeL-1 is more portable than if we generated assembly code.
- Writing a C code generator is easier than writing an assembly code generator. This allowed us to implement a ParCeL-1 compiler in a few years on several parallel systems.
- We can easily take advantage of numerous C libraries available on every system.

6 Benchmarks criteria

In order to evaluate ParCeL-1 performance, we define a sequence of criteria. The first criterion evaluates single-processor performance. Successive criteria give more and more refined information on performance comparison.

6.1 Single-processor performance

As the ParCeL-1 compiler generates C code with parallel library calls, and as C is an efficient and general purpose sequential language, it seems logical to compare single-processor execution of ParCeL-1 programs to the execution of corresponding C programs. Thus, we first measure the overhead of the automatic C code generation of ParCeL-1 compared to a manually implemented C program. We define the Slow-Down of level 0 as follows:

$$SD0 = \frac{T_{\mathsf{ParCeL-1}}(1)}{T_C(1)} \tag{1}$$

where $T_{\mathsf{ParCeL-1}}(1)$ is the time to run a ParCeL-1 program on one processor and $T_C(1)$ is the time to run the best program in C implementing the same algorithm.

6.2 ParCeL-1 implementation scalability

The second stage of our sequence of comparison criteria consists of computing the speed-up of a ParCeL-1 program execution on P identical processors versus the execution on only one of these processors. This criterion measures the efficiency of the ParCeL-1 parallelization. The level-one speed-up is defined as:

$$SU1(P) = \frac{T_{\mathsf{ParCeL-1}}(1)}{T_{\mathsf{ParCeL-1}}(P)} \tag{2}$$

where $T_{\mathsf{ParCeL-1}}(P)$ is the execution time of the benchmarked program on P processors; $SU1$ is a measure of the scalability of a ParCeL-1 program.

6.3 Speed-up of ParCeL-1 programs

At the next stage of our sequence of criteria, we compare the execution time of parallel ParCeL-1 programs on P processors to that of sequential C programs. We define the level-two speed-up as:

$$SU2(P) = \frac{T_C(1)}{T_{\mathsf{ParCeL-1}}(P)} \leq P \tag{3}$$

This speed-up can be expressed as a function of the previous measures, combining equations 1, 2 et 3:

$$SU2(P) = \frac{SU1(P)}{SD0} \tag{4}$$

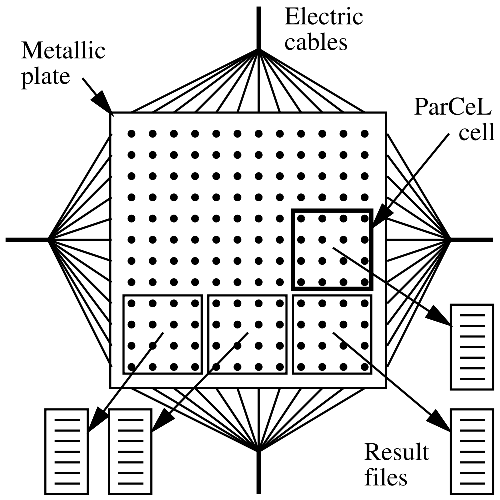


Figure 10: ParCeL-1 Jacobi relaxation implementation

The $SU2$ performance measure is the most relevant one for an application user. An efficiency corresponding to this speed-up can also be defined as follows:

$$E2(P) = \frac{SU2(P)}{P} \tag{5}$$

The following relation that illustrates the slow-down of ParCeL-1 programs versus C programs can be easily derived from equations 3, 4 and 5:

$$E2(P) \leq \frac{1}{SD0} \tag{6}$$

7 Numeric computation applications

In this section we present two computation-intensive programs (Jacobi relaxation and N-body problem) and a tree exploration program (N-queens problem), and we evaluate their parallel implementation. In each case we have measured the SD0, SU1, SU2 and E2 coefficients on a Cray T3D computer.

7.1 Jacobi relaxation

The Jacobi relaxation is an iterative computing algorithm used to solve certain differential equation systems. For example, it can be used to compute the voltage distribution in a metallic plate on which a known potential distribution is applied on the sides (figure 10). The plate is approximated by a grid of points, and at each step of the relaxation the potential of each point is updated by computing the average of its four direct neighbors. In this example, the voltage distribution is computed after approximatively 2000 iterations.

The ParCeL-1 program consists of a main cell that creates a complete network of computing cells. Each of them is in charge of a sub-grid of points and saves its local results in a separate file, as shown in figure 10.

With a grid of 256 $\times$ 256 points spread on 256 cells (256 points per cell) and 1000 relaxation steps, the evolution of the execution time and the SU2 are shown in figure 11. The parallel ParCeL-1 program is faster than the sequential C program, but the speed-up does not exceed 3, even on 16 processors. In fact, the Jacobi relaxation is unfavorable to a parallelization with ParCeL-1. First, grid point computations are very simple, and

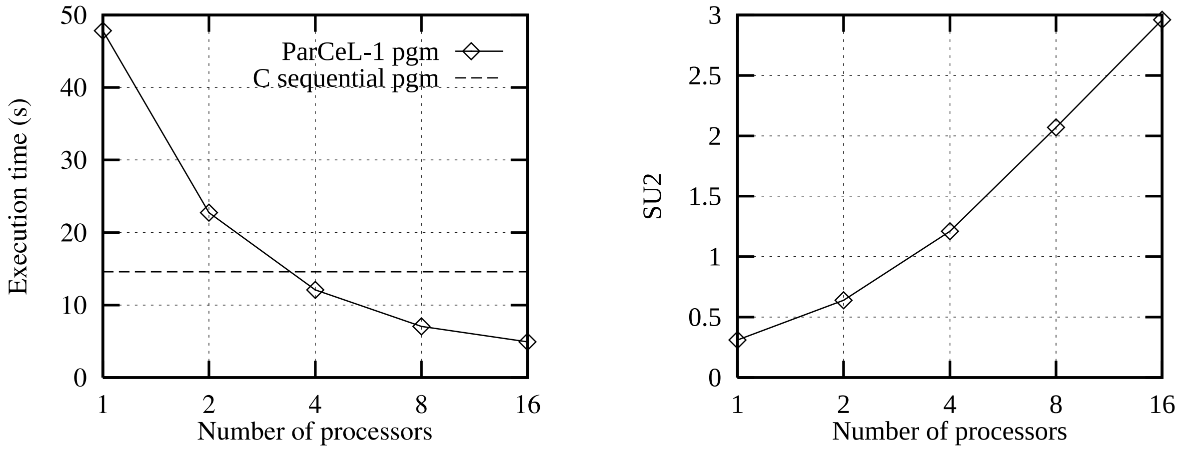


Figure 11: ParCeL-1 Jacobi relaxation execution time and SU2 on a T3D multicomputer

comparatively, the overhead for cell system management is not negligible. Therefore, the sequential slow-down $SD0$ is extremely important in this example (nearly 5), limiting the effects of parallelization. Second, the cell connection pattern only requires local communications with neighbors, while ParCeL-1 implements systematic global communications for any kind of connection patterns. The communication overhead, therefore, limits the speed-up when the number of processors increases.

7.2 N-bodies problem

The N-bodies problem consists in determining the trajectories of N bodies in mutual interaction (e.g., through gravitation), each body being influenced by all the other ones. The computing method is iterative. At each time step, the acceleration of all bodies is computed based on the position of all other bodies and on the law of interaction. An approximation of the speed and then of the displacement of the bodies can be computed, yielding their positions at the next iteration.

As for Jacobi relaxation, the implementation uses a specific cell to create the network of working cells, and each computing cell manages several bodies and local files to store body trajectories. The cells are fully interconnected so that each body can be aware of the position of all the others (figure 12).

For 512 bodies, the performance of the ParCeL-1 program is very satisfactory (figure 13). The speed-up versus a sequential C program ($SU2(P)$) and the corresponding efficiency ($E2(P)$) are presented in figure 14. The sequential slow-down $SD0$ amounts to 1.4 and limits the efficiency to approximately 70%, as shown in figure 14, even for a small number of processors. The efficiency decay for a large number of processors is due to load imbalance, communication overheads and sequential fraction problems (Amdahl's law [3]) that become critical beyond 100 processors. But performance increases with the problem size, as shown in table 2.

7.3 N-queens problem

The well known N-queens problem lies in determining how N queens can be placed on an $N \times N$ chess board so that no queen may attack another given the normal chess rules for moving a queen. If we restrict the problem to the research of the number of solutions, then we have to program an exhaustive tree exploration, and each subtree can be explored independently of the others. As a subtree search remains an important amount

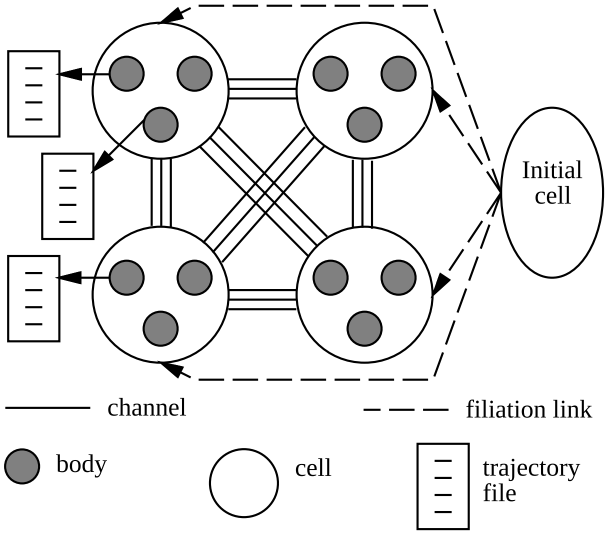


Figure 12: The N bodies implementation

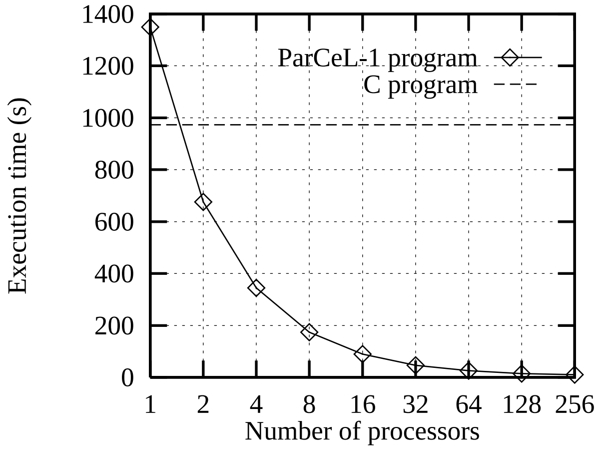


Figure 13: Execution time of the 512-bodies problem on a T3D multicomputer

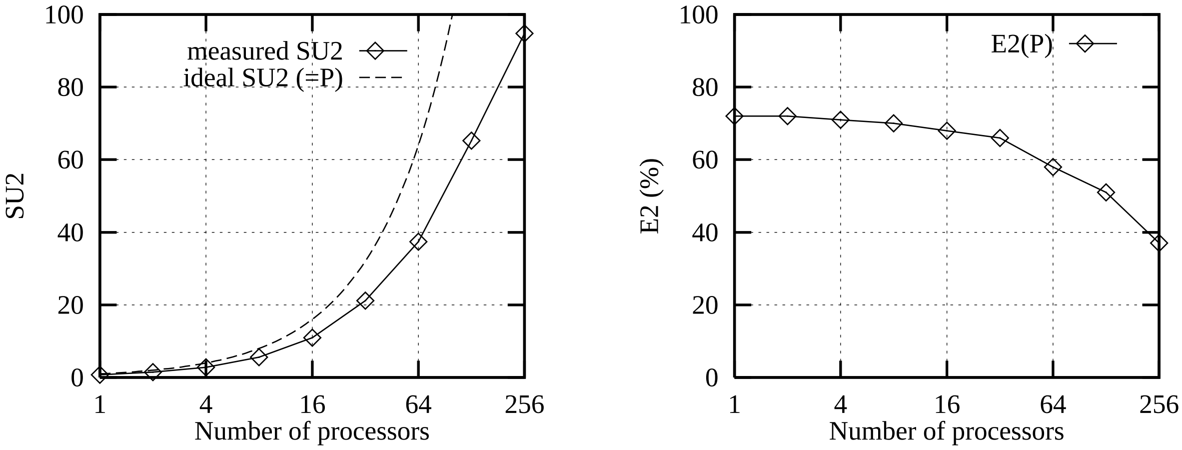


Figure 14: Speed up and efficiency of the 512-bodies problem on a T3D multicomputer

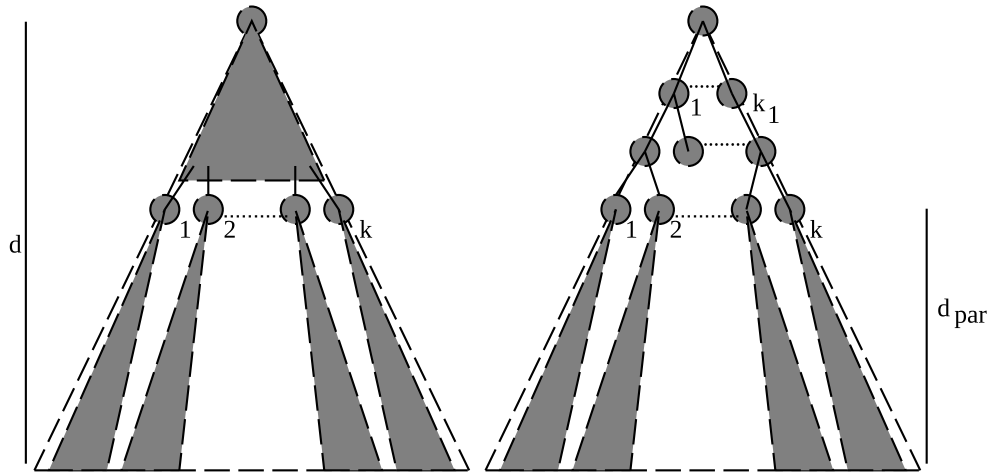


Figure 15: Two ParCeL-1 implementations of the N-queens problem: sequential and parallel splitting

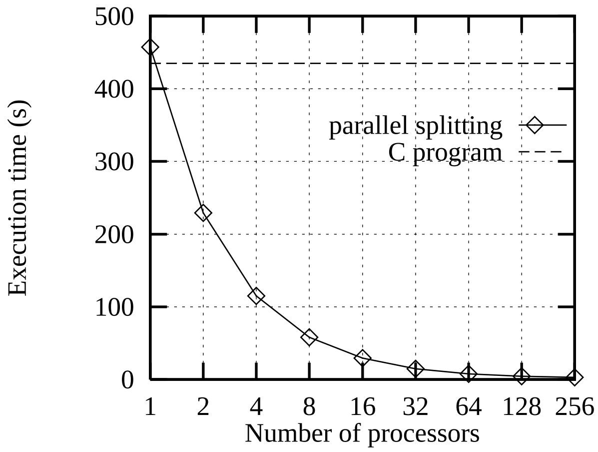


Figure 16: Execution time of the 15-queens implementation with parallel splitting, on a T3D multicomputer

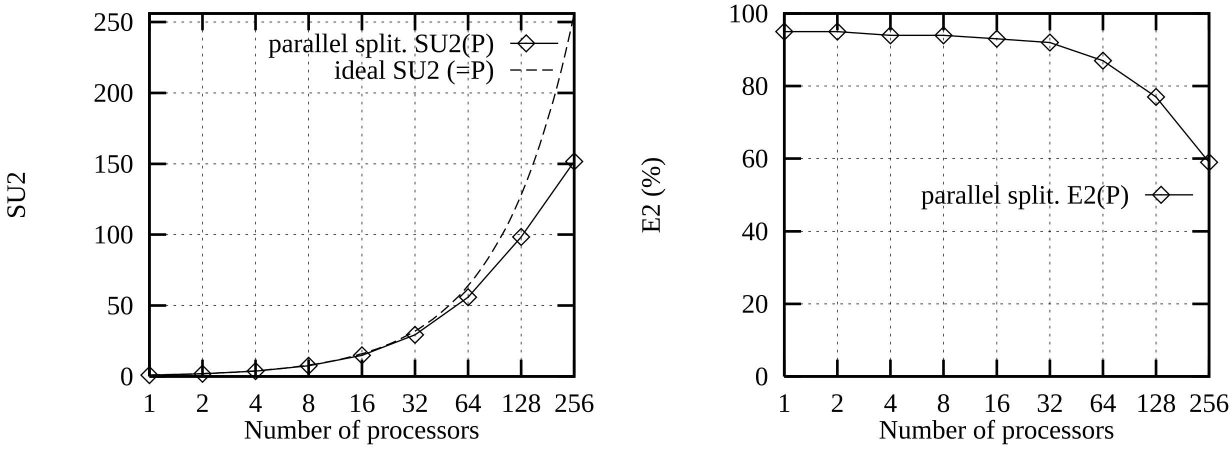
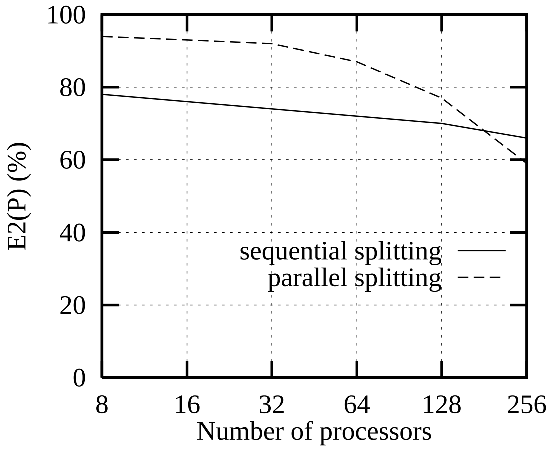


Figure 17: Speed-up and efficiency of the 15-queens implementation with parallel splitting, on a T3D multicomputer

N	Bodies per cell	SU2 (1000 iter)	E2 (1000 iter)
512	2	94.5	37%
1024	2	108.3	42%
1024	4	133.0	53%
2048	4	141.5	55%

Table 2: Performance evolution of the ParCeL-1 N-Bodies program on a T3D multicomputer with 256 PEs when the problem size grows



	P	SU2	E2
seq. splitting	256	168	66%
par. splitting	256	151	59%

Figure 18: Comparison of efficiency of the two 15-queens implementations, on a T3D multicomputer

of computation and as all subtree search are different, this problem is well adapted to control parallelism with medium grain programming. We just have to split the initial tree exploration into numerous subtree explorations, to execute those in parallel, and to sum up the partial results from all subtrees. We obtained a speed-up of up to 168 compared to a sequential C program ($SU2$), on 256 processors of a T3D, for 15 queens. This performance is very good and better than that of the N-bodies problem.

We programmed two implementations of this problem, both dynamically splitting and pruning the search tree of depth d. They are illustrated in figure 15. The first implementation does the splitting sequentially: a single cell splits the initial tree in k sub-trees of depth d_{par} and creates k new cells in order to search the k sub-trees in parallel. In the second implementation, one cell splits the initial tree in k_1 sub-trees of depth $d-1$, and creates k_1 new cells to manage these sub-trees in parallel. These k_1 new cells run in parallel, and split their sub-trees in new sub-trees of depth $d-2$. The algorithm iterates until depth d_{par}, which supplies k sub-trees as in the sequential splitting program.

The execution time, the speed up and the efficiency versus C program of the parallel splitting version are given in figures 16 and 17. Results are very good, but figure 18 shows that the sequential splitting version has better efficiency on 256 processors. In fact, the sequential splitting program has a more important sequential slow-down ($SD0$) than the parallel splitting program, whose sequential parts have been optimized. It is, therefore, not as efficient as the parallel splitting version for a small number of processors. However, it scales better than the more sophisticated implementation and exhibits a better performance on 256 processors. This phenomenon may be explained by the fact that the parallel split uses more cells and channels, and therefore spends more time managing them.

Load balancing is also more difficult for this version, because the dynamic exploration and pruning creates a great number of cells irregularly over several cycles. These problems become critical for a large number of processors.

8 Artificial intelligence related applications

In this section we present two AI related applications written in ParCeL-1: the well known Kohonen neural network, and the hybrid symbolic-connectionist system Resyn we designed.

8.1 Kohonen self-organizing feature map

The Kohonen self-organizing map [15] is the most well known among neural networks using unsupervised training. Its applications are in the field of vector quantization and mapping of multi-dimensional spaces. A Kohonen map is a network of N neurons, usually arranged on a two-dimensional grid. To each neuron is associated a weight vector whose length is the number of inputs of the neural network. Training is performed iteratively. At each iteration t, an input vector is chosen randomly in the training set. For each input, its distance from each neuron's weight vector is measured. Then the neuron with the smallest distance (the winner neuron) is determined and all weights are updated according to a given learning law.

The ParCeL-1 implementation of a Kohonen self-organizing map uses two types of cells (fig. 19): many cells of type `neuron group` and one supervising cell. The neuron grid is divided among the `neuron group` cells. The supervising cell first creates the `neuron group` cells. It is then responsible for inputing training vectors to the neuron mesh, getting back the corresponding neuron activations, finding the minimal output value (and thus the winner), and broadcasting to every neuron group the identity of the winner neuron.

With a network of 1600 neurons spread over 64 neuron groups (25 neurons per neuron group), 4 dimensional randomly generated vectors and 5000 iterations, we got the results presented in figures 20 and 21. The results are fair (with a speed-up reaching 5.4 on 32 processors, showing the interest of the parallelization of the network). The Kohonen program involves more communication than the N-Queen problem and less computation than the N-bodies problem, and its performance is not as good as any of those. On the contrary, it involves more computation than the Jacobi program, and has better performances than that program.

8.2 An hybrid system implemented in ParCeL-1

An example of a hybrid symbolic-connectionist system, called Resyn, has been implemented in ParCeL-1. Resyn is a network of various types of units exchanging both numeric and symbolic information. Its user interface is an interpreter of a dedicated language used to build or query the network. An earlier version of Resyn is described in more detail in [16]. Resyn is used to represent common sense knowledge and gives a framework for analogy making. Figure 22 shows the basic principles of Resyn and its ParCeL-1 implementation.

The Resyn network is constituted of two kinds of nodes: nodes representing objects, and nodes representing properties. Resyn is a *localist* network [18]: each node has a given meaning. Two different inference mechanisms are implemented: a symbolic marker-passing mechanism to represent symbolic assertions such as *shark is a fish* or *dolphin is a mammal*, and a numeric activation spreading mechanism for uncertain information. Links

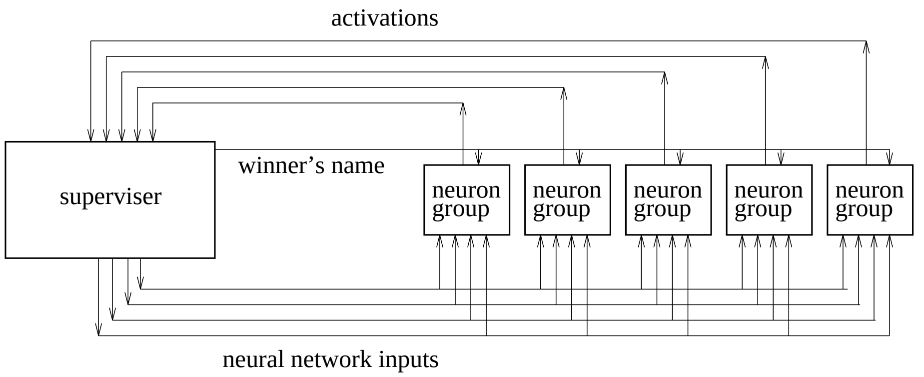


Figure 19: Kohonen feature maps in ParCeL-1: implementation principle

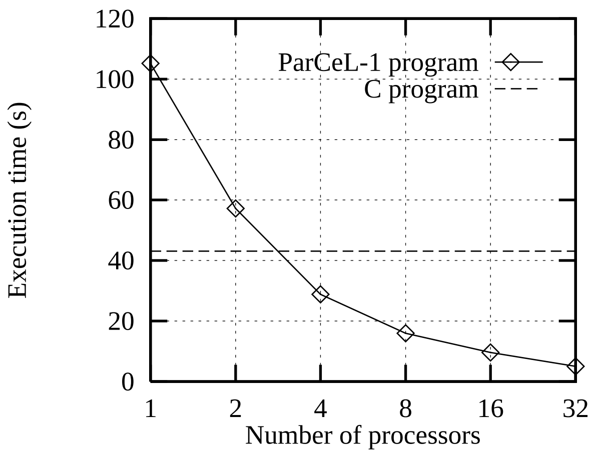


Figure 20: Execution time of the Kohonen map implementation on a T3D multicomputer

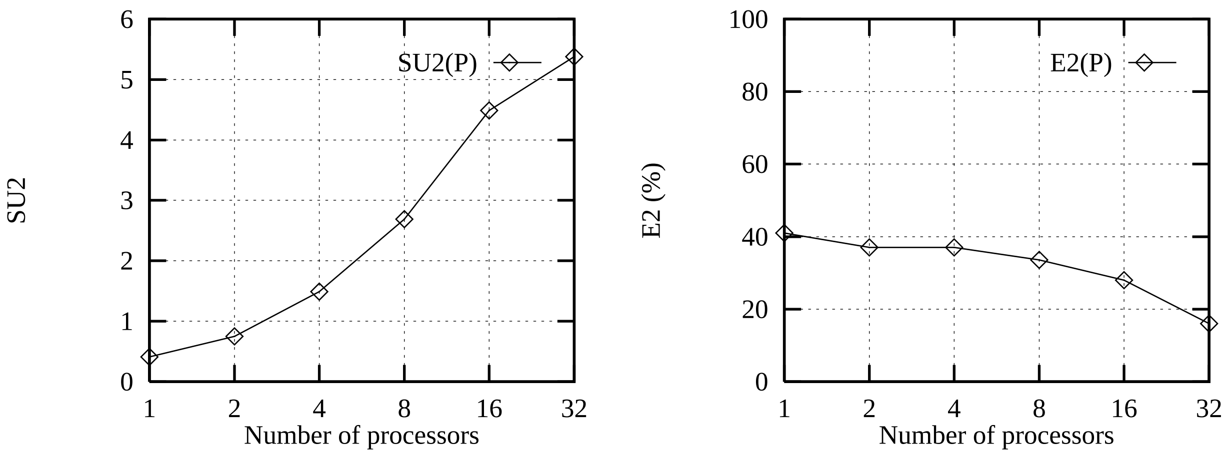


Figure 21: Speed-up and efficiency of the Kohonen map implementation on a T3D multicomputer

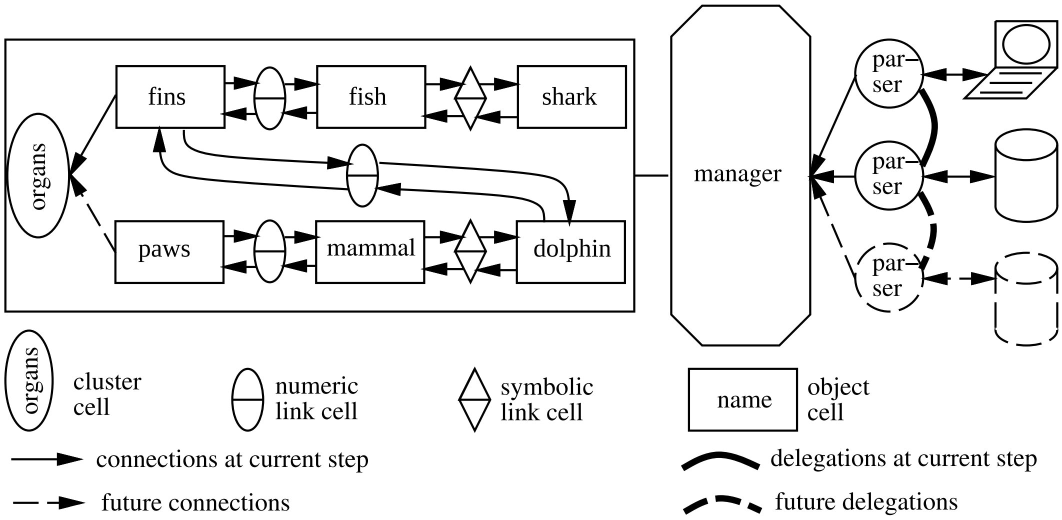


Figure 22: Resyn network topology.

carrying these activations are weighted according to their evidence: for example, there will be a strong link between *mammal* and *paws*, meaning *mammal often have paws*. Nodes representing properties are part of a cluster enforcing mutual inhibition: the nodes *fins* and *paws* will be part of the cluster *organs*. The network can be instructed that an object is like another one, and will then modify itself to integrate this new information: the first object will inherit the properties of the second one, but will keep its *is_a* hierarchy. In the example network represented in figure 22, the numeric link between *dolphin* and *fins* has been added by the network to integrate the statement *is_like(dolphin, fish)*. A query on *dolphin* to the network will return that a dolphin is a mammal and that it has fins.

The network is queried by giving some original markers and activations to the chosen nodes. Several propagation cycles are accorded to the network to relax and to get to a steady state. The network answer is a list of activated nodes representing the relevant concepts and their properties. Due to its hybrid nature, the model can represent hierarchies, but also analogies between objects, not unlike [24]. If several objects share enough common features, the activation of one of them will cause the activation of the other ones. In a similar manner, when some features of an object are active enough, then this object will tend to become active too.

The ParCeL-1 implementation of an heterogeneous network such as Resyn uses several possible programming methods. Each node and each link of the network is naturally mapped on a cell in the program. This network evolves during the relaxation phase as new links are created. The `manager` cell is implemented as an automaton: a variable containing the current state of the automaton is used in the condition parts of the rules. This cell interprets the user's commands and executes the corresponding actions, especially creation and connection of nodes in the network. It receives its commands via a parser cell. Initially, a keyboard `parser` cell is created. If the user requests a file load, then this cell delegates the file parsing to a specialized cell. This delegation mechanism, following Aghas's model [1], is transparent to the `manager` cell. Communications between `manager` and `parser` cells are implemented in an asynchronous manner using two additional coordination channels. This protocol prevents any message loss and keeps the network from being blocked by a wait for a keyboard input.

Parallel performance of Resyn has not been measured; on small exemplar data bases, the computation times are too small to be significant, and at this point we do not dispose of a big enough database to compile into Resyn. However, this type of algorithm is by essence parallel, and we expect that for a big enough network, the execution of Resyn on a parallel computer would be profitable.

9 Conclusion

In this article, we presented a complete language (parser - compiler and libraries) which runs on several parallel systems: Telmat T-Node, Intel Paragon, Cray T3D, SGI Power Challenge and a network of workstations with PVM. The same ParCeL-1 source code can be compiled and executed on any of these architectures, for any number of available processors. ParCeL-1 has an original computing model based on autonomous and active actors which can compute at each cycle, regardless of communications. This originality leads to specific consequences for two key aspects of any parallel programming language: the semantics and ease of programming on the one hand, and the parallel performance on the other hand. For each aspect, we summarize what ParCeL-1 offers and what it lacks, and we propose new ways to improve it.

9.1 Semantics and ease of programming issues

In ParCeL-1, each actor is active like a process. This feature increases parallelism in program conception because it leads programmers to design their implementation as a network of active processes, computing in parallel. Some small programs (e.g. N-queens) have been written efficiently in a short time by students, indicating that one can easily master ParCeL-1 and its computational model. The semantics of ParCeL-1 make it intuitive to use to implement networks composed of small computing elements—such as neural or semantic networks, where the computing elements can be naturally mapped on cells. ParCeL-1's versatility and parallel implementation make it especially attractive as a connectionist language. However, as the examples presented in this article show, ParCeL-1 is not restricted to that type of applications.

Through our and other users' experience in programming ParCeL-1[17, 29], it appeared that the language still lacks of flexibility and expressiveness in some circumstances. We identified the following limitations:

- Only one piece of data can be sent on a channel at each cycle (no queuing system is associated to a channel). This makes data management tedious when the number of messages to be sent to a cell at each cycle is not known at compilation time.
- No communication can take place between two cells without prior declaration of a communication channel, and no shared memory is available between cells. This can be a drawback when the communication pattern in the application is unpredictable, for instance when all cells need to randomly access a common database.
- Dynamic memory allocation is only available through the creation of new cells, not inside a cell or in the shared space of communication channels. This makes recursive data structures such as lists and trees difficult to implement.

We are currently beginning to design a new version of the ParCeL-1 language for artificial intelligence and connectionist applications. We aim at developing a language for modern parallel architectures and environments, fully using their capabilities, with higher-level semantic than ParCeL-1 and making the parallel programming of our applications easier. This new version will plainly take advantage of the shared-memory functionalities

Application	Processors	Speed up *vs* C program
Jacobi relaxation	16	3
Kohonen map	32	5.4
2048-bodies problem	256	141
15-queens problem	256	168

Table 3: Best speed-ups (SU2) obtained with several test programs on a T3D multicomputer

of recent parallel architectures. It will feature message queues and a global shared dataspace.

Two new PhD theses have begun in 1996 and 1997 at the LORIA (Nancy, France) in collaboration with SUPELEC (Metz, France). They investigate new ways to easily parallelize connectionist and multi-agent systems. Part of their results will be used to better design a new version of the ParCeL-1 language for artificial intelligence and connectionist applications.

9.2 Performance issues

We have tested ParCeL-1 with various benchmark programs and we obtained fair to very good results. The best results for each program are summarized in table 3, showing here again that the computational model is not restricted to a specific kind of application. The parallel performances offered by ParCeL-1 seem to be a good compromise between an automatic parallelization of the source code, still out of reach, and a lower-level, architecture-dependant parallel programming. A low-level language may give better results in terms of pure performance, but the masking of the parallel architecture and of the communication layers make ParCeL-1 suitable for a quicker and portable program development.

However, here again, ParCeL-1 presents some limitations. As shown in the analysis of the Jacobi relaxation example (section 7.1), there is a practical minimum to the amount of computation a cell ought to perform at each cycle. If to little computation is performed by each cell, the cost of cell management and system house-keeping becomes prohibitive in comparison. Improving the code generation of the current ParCeL-1 language could lead to some performance improvement; for instance, it should be possible to better take advantage of the global address space of recent multi-processor architectures (global shared memory for the SGI O2000, remote memory access for the Cray T3E). However, the lack of performance of ParCeL-1 for small cells is mainly due to the semantics of the language itself. The computational model is completely dynamic; creation and destruction of cells and channels may occur at any time and anywhere in a program. This freedom prevents a lot of optimization during the code generation.

For this reason, we are currently investigating the introduction in the language of static networks of cells, declared at compilation time. The following benefits are expected from this new feature:

- Direct access by the cells to input and output channels (it currently involves two levels of indirection).
- Communication in one step only, instead of the three sub-steps described in section 5.
- Suppression of garbage collection in the channel tables.
- Communication of output channels restricted only to the remote processors who need it.

- Possible inlining of the the update procedure for small cells.

Extending the ParCeL-1 model with static cell networks requires new primitives to specify these networks. A new PhD thesis was started on this topic in 1997 at the Swiss Federal Institute of Technology in Lausanne.

9.3 Synthesis

The different improvements discussed above—improvement of the expressive power on the one hand, and of the performance on the other hand—lead to extending the current ParCeL-1 model in two opposite directions. The addition of message queues and a shared data space will make the language more complex and less easy to compile. Conversely, static networks of cells will not contribute to an easier programming of complex data structures.

The main challenge for the future of ParCeL-1 is to investigate how static networks and networks with increased dynamicity could be mixed seamlessly in a unique programming model. As with any other programming model, improving both the efficiency and the programming flexibility will involve tradeoffs that remain to be determined.

Acknowledgments

We thank Niels Jacobsen, Peter Lewis and Claude Lhermitte for their helpful comments. We also thank the Swiss Federal Institute of Technology (Lausanne, Switzerland) for giving us access to their Cray T3D computer, and the Centre Charles Hermite (Nancy, France) for giving us access to their Silicon Graphics Power Challenge Array. This research is supported in part by the ONR, Contract Number N00014-89-J-1975N158. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ONR or the US Government. The two PhD theses currently going on at LORIA are both funded by the Centre Charles Hermite and the Université Henri Poincarré (Nancy, France).

References

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

[2] N. Almassy, M. Köhle, and F. Schönbauer. Condela-3: A language for neural networks. In *International Joint Congress on Neural Networks*, pages 285–290, San Diego, 1990.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings*, Atlantic City, N.J., 1967.

[4] A. S. Bavam. Nps: A neural network programming system. In *International Joint Congress on Neural Networks*, pages 143–148, San Diego, 1990.

[5] J. P. Briot and R. Guerraoui. A classification of various approaches for object based parallel and distributed programming. Technical report, University of Tokyo and Swiss Federal Institute of Technology, 1996.

[6] A. Burns. *Programming in Occam-2.* Addison Wesley, 1988.

[7] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing — a paradigm for transportable software. In *Proceedings of the 28th*

Annual Hawaii Conference on System Sciences, volume II. IEEE Computer Society Press, January 1995.

[8] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. IC++: A C++ dialect for high performance parallel computing. In *2nd international symposium on object technologies for advanced software (ISOTAS)*, Kanazawa, Japan, March 1996.

[9] Thierry Cornu and Stéphane Vialle. A framework for implementing highly parallel applications on MIMD architectures. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, Oxford Science Publications, pages 314–337. Oxford University Press, 1995.

[10] M. W. Goudreau, S. B. Rao, and T. Tsantilas. A study of the BSP model: Algorithms and implementation. Technical Report UCFCS:CS-TR-94-04, Department of Computer Science, University of Central Florida, April 1994.

[11] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *Computer*, 26(5), May 1993.

[12] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *International joint conference on artificial intelligence*, 1973.

[13] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[14] W. Kim and G. Agha. *Compilation of a highly parallel actor-based language*, pages 1–15. Lecture notes in computer science 757. Springer-Verlag, 1993.

[15] T. Kohonen. *Self-Organization and Associative Memory*, volume 8 of *Springer Series in Information Sciences.* Springer-Verlag, 1989.

[16] Y. Lallement, T. Cornu, and S. Vialle. An abstract machine for implementing connectionnist and hybrid systems on multi-processor architectures. In V. Kumar H. Kitano and C. B. Suttner, editors, *Parallel Processing for Artificial Intelligence, 2.* Elsevier Science Publishers, 1994.

[17] Y. Lallement, Th. Cornu, and S. Vialle. Application development under parcel-1. In H. Kitano, C. Suttner, and J. Geller, editors, *Parallel Processing for Artificial Intelligence, 3.* Elsevier Science Publisher, 1997.

[18] Y. Lallement, M. Hilario, and F. Alexandre. Neurosymbolic integration: Cognitive grounds and computational strategies. In M. DeGlas and Z. Pawlak, editors, *World Conference on the Fundamentals of Artificial Intelligence*, Paris, July 1995.

[19] R. R. Leighton. The aspirin/migraines neural network software, user's manual. Technical Report MP-91W00050, MITRE Corporation, 1992.

[20] R. Miller. A library for bulk-synchronous parallel programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.

[21] J. Pallas and D. Ungar. Multiprocessor Smalltalk: A case study of a multiprocessor-based programming environment. In *Conference on programming language design and implementation*, pages 268–277, Atlanta, June 1988.

[22] L. Prechelt. Cupit — a parallel language for neural algorithms: Language reference and turotial. Technical report, Uni. Karlsruhe, Germany, 1994.

[23] S. J. Rogers. Neudl: Neural-network description language. Available by ftp at cs.ua.edu, file /pub/neudl/neuDLver0.2.tar.gz, August 1993.

[24] R. Sun. *Integrating Rules and Connectionism for Robust Commonsense Reasonning.* John Wiley and Sons, New York, NY, 1994.

[25] D. G. Theriault. Issues in the design and implementation of act 2. Technical Report AI-TR-728, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1983.

[26] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

[27] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18. Elsevier Science Publishers, 1990.

[28] S. Vialle, T. Cornu, and Y. Lallement. Parcel-1, user's guide and reference manual. Technical Report R-10, Supelec Metz campus, SUPÉLEC, 2 rue Edouard Belin, F-57078 Metz Cedex 3, November 1994.

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

[30] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura. Implementing concurrent object-oriented languages on multicomputers. *IEEE parallel and distributed technology*, 1(2):49–61, May 1993.