ParCeL-1
User's guide & reference manual

ParCeL-1 version 1.4.1

Manual version 1.4.0

Stéphane Vialle
SUPÉLEC[1] & PRiSM[2] & CCH[3], vialle@ese-metz.fr

Thierry Cornu
EPFL DI[4], cornu@di.epfl.ch

Yannick Lallement
CRIN-INRIA Lorraine[5] & CCH, lallemen@loria.fr

SUPÉLEC Metz Campus technical report: R-10
CRIN-INRIA Lorraine technical report: 94-R-235

August 10, 1995

[1]SUPÉLEC, Technopôle Metz 2000, 2 rue Edouard Belin, F-57078 Metz Cedex 3

[2]PRiSM, Université de Versailles St-Quentin, 45 Avenue des Etats-Unis, F-78035 Versailles Cedex

[3]Centre Charles Hermite, Campus Scientifique, Bâtiment Loria, BP 239, F-54506 Vandœuvre-lès-Nancy

[4]Swiss Federal Institute of Technology, EPFL DI, Mantra Group, CH-1015 Lausanne

[5]CRIN-INRIA Lorraine, Campus scientifique, Bâtiment Loria, BP 239, F-54506 Vandœuvre-lès-Nancy

Contents

List of Figures

Chapter 1

General overview

1.1 Introduction

ParCeL-1 is a parallel programming system using synchronous concurrent objects called cells. This system is dedicated to explicitly distributed applications with a medium or fine grain of parallelism. Applications requiring a high density of communication are well supported. Neural networks, simulation of digital machines, distributed artificial intelligence are typical target applications. The ParCeL-1 system was designed with the following objectives:

- ability to execute on several multi-processor architectures;
- ability to hide from the programmer most of the implementation constraints, especially the actual number of processors and their arrangement;
- ability to convey a precise idea for the programmer of how computation will be achieved and parallelized.

ParCeL-1 is implemented on a virtual machine, called the Virtual Cellular Machine (VCM-1) which takes care of process creation and placement, and of inter-process communication. Table 1.1 shows the list of machines that currently support the ParCeL-1 system.

Unix based uniprocessor	**MIMD machines**
DEC Alpha	Telmat T-node
SunOS SparcStation	Parsytec mtmsun board
Linux PC compatible	Cray T3D
HP 9000 S 700	

Table 1.1: ParCeL-1 supported architectures

1.2 The cellular computational model

1.2.1 Overview of the cellular network principles

The underlying computational model of the cellular model is an agent-based model. It belongs to the family of actor- and object-based models. Its basic agents are called cells. A cellular program consists of a network of cells computing in parallel. The main features of the computational model are:

- The communication graph between cells is explicit. This means that a connection between two agents must be declared explicitly and lasts until explicitly destroyed. This is the primary difference from the classical actor models in which messages are sent using logical addresses, thus providing abstraction for the communication pattern. Explicit declaration of connections was a deliberate choice motivated by two distinct reasons. First, most target applications of ParCeL-1 (especially neural nets) exhibit an explicit communication pattern. The form of this pattern is usually important to characterize the algorithms. Second, declaring communications explicitly gives the ParCeL-1 programmer an idea of the communication burden implied by his programming style.

- The users declare cell types statically before run time. Thus cell types may be compiled in a simple way. Since no new cell type can be created at run time, the model avoids dynamic compiling, as well as migration of code in the case of distributed architectures.

- The cells of a ParCeL-1 program are dynamically created instances of the static pre-declared types. The pattern of connections between cells may also be created and modified dynamically. Dynamic configuration is a major feature of the cellular model. Most of the ParCeL-1 potential applications require this feature. This is the case for semantic networks and for several neural net models.

- Finally, the main feature of the cellular model is that it is globally synchronized. Every cell updates synchronously in a cyclic way, making the cellular model a synchronous MIMD model. This is probably the main difference from classical actor models of computation where concurrency is asynchronous. The effect of synchronism is to separate computation from communication in the model.

1.2.2 Concept of a cell

A cell may be seen as a basic abstract processor, with internal data storage, explicit connections to other cells, and an associated executable program

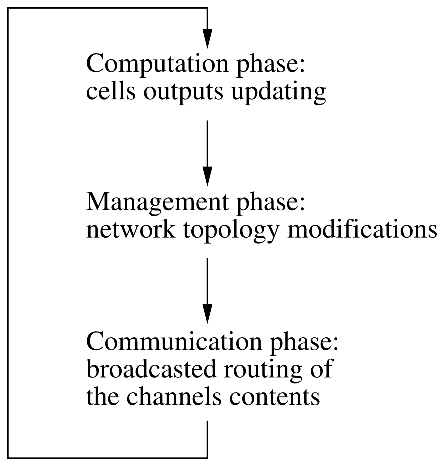

Figure 1.1: Main stages of a ParCeL-1 cycle

responsible for periodically updating the output values of the connections to other cells.

A ParCeL-1 program consists of a list of cell type declarations of which the actual cells of a program will be instances. These declarations include for each cell type the following elements:

- The cell type data structures, including the private variables names and those of the input and output channels to connect to other cells. The same output channel may be connected to an unlimited number of input channels. Thus the communication pattern may be more general than just point to point connections.

- A set of transition rules specifying the actions to perform during each update, associated with each cell type. A transition rule consists of a precondition (head of the rule) and an action (body of the rule). A precondition is a boolean expression testing the value of input channels and private variables. An action is a short sequential program. In ParCeL-1this program is expressed in a classical procedural style.

A cell update consists of a matching for all transition rules and a firing of at most one rule body.

1.2.3 Basic cycle of a cellular network

Figure 1.1 gives the successive phases of a cellular cycle. Each cycle is composed of three phases:

- The first phase is a computation phase, during which the outputs of the cells are updated according to the former values of their inputs and private variables. During this phase, cells may also issue requests for creation of new cells thus establishing and modifying connections between them. Upon creation, each cell receives a unique identifier. This identifier serves as a logical name for the cell and will be used for further connection requests.

- The second phase is a management phase. During this phase, the requests for dynamic changes in the cellular topology are taken into account.
- During the last phase (the communication phase), the newly computed output of the communication channels between cells is sent to the receiving cells.

When a ParCeL-1 program executes, only one cell, of a type called *main*, is initially created. It is up to the user to define this cell type. The initial cell will be the predecessor of every other cell of the program.

1.3 Overview of this manual

This manual is divided into three main sections:

- The user's guide describes the language and its features. This section contains the basic concepts of ParCeL-1 and some examples of programs. Thus a beginner should read this section first to learn the use of the language. Every ParCeL-1 keyword or operator presented in the user's guide is further detailed in the reference manual.
- The reference manual contains a complete alphabetic list of the ParCeL-1 types, operators, requests, and instructions.
- The libraries reference manual contains a complete alphabetic list of the ParCeL-1 system libraries, requests and functions.
  In the last two chapters, the nature, syntax, description, and some examples of use for each item appear on one page.

The first appendix deals with the installation of the ParCeL-1 compiler on the various supported architectures. The second appendix gives a comprehensive list of the ParCeL-1 error messages. Finally, a glossary gives a short definition of every new ParCeL-1 concept.

Should you discover a problem, please report it by electronic mail to *vialle@ese-metz.fr* stating the circumstances of the problem.

Chapter 2

User's guide

2.1 An introduction

This manual introduces the ParCeL-1 language. The reference manual describing each ParCeL-1 instruction is located in the next chapter. Readers of this reference manual are assumed to have a basic knowledge of the C language and programming. The ParCeL-1 compiler and syntax are similar to the C compiler and syntax. One of the main differences between C and ParCeL-1 is that ParCeL-1's instructions return no value. For example, in the case of the assignment (`=`) it is thus impossible to write `x = y = 0`, it is necessary to write `x = 0; y = 0`. In this manual, we will emphasize the common points with C but not present further detail.

2.1.1 First example: printing *Hello world!*

Most programming language user's guides begin with the "Hello world!" program as an example. Let us examine this program in ParCeL-1:

```
# include <hcifunc.h>

typecell main() {

  ==> {prints('Hello ');
       prints('world!');
       printnl();
       halt;
     }
}
```

As in C, the header files containing the definitions of the functions used in the program must be included. Here the `prints` function is used, which belongs to the hcifunc library. Thus `#include <hcifunc.h>` is necessary.

The string *Hello world!* cannot be printed all at once, since ParCeL-1 does not handle the *string* type. Printed and manipulated objects are called *symbols* and are 0 to 8 characters long only. Thus `prints` stands for *print symbol*.

Finally, the programmer has to specify the end of the program by `halt`. Since a ParCeL-1 program is cyclic, *Hello world!* would be printed forever without the `halt` instruction.

Warning: the input/output instructions used here, namely `prints` and `printnl` are part of the hcifunc library (see 2.2.10). This library is *not* available on every target architecture (see A.4 for details).

2.1.2 Second example: prime numbers

The following program gives some examples of cell creation and connection. It declares two types of cells: `main` and `filter`.

```
#include <hcifunc.h>

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

  ==> {                        /* 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. */
  immat filter f;               /* an immatriculation 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 += 1;            /* increase new_int at each cycle.      */
}
```

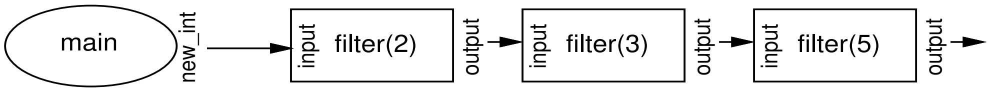

Figure 2.1: Communication graph of the prime numbers program

This program is used to generate prime numbers. The basic principle of the algorithm is to use a series of filters, each filter being initialized by a prime number, and filtering each number divisible by that prime.

The program listed above works as follows: a `main` cell is created when the program starts. It is used as a generator of integers (called `new_int` in the program). A first filter (the cell type `filter` in the program) is created in the `initially` rule of the `main` cell (line: `f = create filter(2)`), which will filter any integer divisible by 2. This filter will let odd numbers pass. The first non filtered number is three, thus three is a prime number, and a new filter is created that will filter all the numbers divisible by three. And the process goes on: 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 points to be noticed in that program are: there are as many `filter` cells as we want, for only one cell type. The `filter` cells create some other `filter` cells by themselves (line: `next = create filter(input)`). The cells communicate through channels connected with one another: see figure 2.1. The output channel `new_int` of the `main` cell is connected to the input channel `input` of the first `filter` cell. Then, the output channel `output` of each `filter` cell is connected to the input channel `input` of the next `filter` cell.

2.2 ParCeL-1 language

2.2.1 Program organization

A ParCeL-1 program is composed of:

- A set of CPP preprocessor instructions (see 2.5.2).
- A set of function and cell prototype declarations (see 2.2.8).
- A set of global function definitions. Each function declared outside a cell type will be usable in each cell type.
- A set of cell type definitions.

A cell type definition is composed of (see section 2.2.6 for more information):

- A header indicating the parameters of the cell type.
- A set of input and output channels.
- A set of internal variables (due to the distributed and parallel nature of the language, there are no global variables in ParCeL-1).
- A set of local functions. Each function declared in a given cell type will be usable only in this cell type.
- A set of *condition* $\Rightarrow$ *actions* rules determining the behavior of the cell.

When a program executes, as in C, one `main` cell is created. This cell is in charge of creating all the others. In any ParCeL-1 program, a `main` cell type must be declared.

2.2.2 Instructions *vs.* requests

Since the language is intrinsically parallel (different cells can be hosted by different processors), two types of statements can be distinguished in ParCeL-1: *instructions* and *requests*.

The instructions are like any other language's instructions: they are executed instantly. An `if` or a `for` will be executed in only one cycle, without any more delay. ParCeL-1 instructions are given in section 2.2.3.

The requests are slightly different in that a request will be executed *after* one cycle. The `create` request, for example, will create a new cell that will only exist and run *in the next cycle*, that is, when the current rule has been totally executed, and another one (or the same) is selected for execution. ParCeL-1 requests are given in section 2.2.3.

2.2.3 ParCeL-1 language components

Keywords

ParCeL-1's keywords are given below. These are only the *basic* language keywords, the functions provided by the libraries are not included in this list.

Instructions			
for	if	else	do
while	return		

Special functions			
connected	free		

Built-in and in-line functions			
exp	exp2	exp10	log
log2	log10	sin	cos
tan	asin	acos	atan
atan2	sinh	cosh	tanh
ceil	floor	rand	iceil
ifloor	srand	fabs	pow
sqrt	fmod		

Requests			
create	connect	connectiffree	disconnect
kill	sleep	awake	halt

Type declaration			
int	double	boolean	symbol
immat	in	out	void
typecell			

Constants			
TRUE	FALSE	NIL	NULL

Private variables			
self	initially	finally	

Punctuation

;	instruction delimiter
,	assignment separator, only used in `for` loops
{}	block marking
( )	boolean or arithmetic expressions parenthesis
[ ]	access to an array
==>	condition / action rule delimiter
-->	subrule delimiter
-(i)->	parametrable subrule delimiter
/* */	comment delimiters
' '	symbol delimiter

Operators

In spite of obvious similarities with the syntax of C, there are no increment/decrement operators (`++`,`--`) in ParCeL-1.

`+ - * / %`	usual arithmetics
`idiv ddiv`	integer and floating point division
`=`	assignment
`+= -= *= /= /= %=` idiv= ddiv=	complex assignment
`== != < > <= >=`	comparisons
`|| &&`	boolean *or*, boolean *and*

Identifiers

Identifiers in ParCeL-1 follow the same rules as in C. An identifier cannot start with a digit. ParCeL-1 is case sensitive, thus the identifiers `a2` and `A2` are different.

2.2.4 Variable types and declarations

ParCeL-1 handles several data types: scalar, derived and channel. The scalar types are `symbol, int, double, immat`. The derived types are the arrays. The channel types are input and output channels (`in, out`).

ParCeL-1 is a strongly typed language. The type of a function or a cell is never attributed by default and must always be known when this function or cell is used. If a functions' type is unknown ParCeL-1 does not assume it to be integer. An attempt to use any unknown function will produce an error.

Scalar types

- `symbol` — a symbol is a string of length 0 to 8 characters. The null value of the symbol type is `NIL`, or `''`. A valid declaration would be `symbol s`.
- `int` — the integer type. The null element is 0. A valid declaration would be `int i`.
- `double` — the floating point numerical type. The null element is 0 or 0.0. A valid declaration would be `double x`.
- `boolean` — a boolean variable can only have two values: `TRUE` and `FALSE`. A valid declaration would be `boolean b`.
- `immat` — the cell immatriculation type. Every cell receives on creation an immatriculation that will be used to refer it in the future. An `immat` must be itself typed by a given cell type. For example, a valid

declaration would be `immat main mi`. Note that one special variable of type `immat` always exists: `self`, pointing out the cell in which it is used. See section 2.2.11 for examples of use.

Arrays

ParCeL-1 can handle arrays. An array is declared by an identifier followed by one or more integer expressions between square brackets indicating the size of the table. For example, `int i[10][5]` will declare a two dimensional table `i`. As in C, the first index of a table is 0, thus the last index is the size minus one.

The type of a table can be any scalar type, including an `immat` type. Thus the definition `immat main m[10]` is valid and can be used to keep the immatriculations of 10 samples of `main` cells.

Channel types

Each cell can have a set of input and output channels, and each channel can have any scalar type. When declaring a channel, it is necessary to declare its type too. An input channel is indicated by the keyword `in` and an output channel is indicated by the keyword `out`. Some valid definitions for a channel would thus be `in int i` or `out immat main im[10]`.

Summary and more examples

- `int i` — definition of the integer variable `i`.
- `symbol s` — definition of the symbol variable `s`.
- `int i[10]` — definition of an array `i` of 10 integers. The indexes range from 0 to 9.
- `immat object im_ob` — definition of the immatriculation variable `im_ob`. This immatriculation is the one of an `object` type cell.
- `in int i` — definition of the integer input channel `i`.
- `in int i[10]` — definition of an array `i` of 10 integer input channels.
- `in immat object p` — definition of the input channel `p` that will contain an immatriculation of an `object` cell type.
- `out immat object p[10]` — definition of an array `p` of 10 output channels, each one containing an `object` cell type immatriculation.

2.2.5 Function declaration and definition

The function definition syntax is the same as in C. A function can be declared either inside a cell (before the rules) or outside any cell. In the first case, the scope of the function will be the cell only, in the second case, the scope will be the whole program, that is the whole set of cell types (see 2.2.7). The types of the parameters must be scalar, not array. A function must be given a scalar type, or `void` if it returns no value. A function can be recursive, that is, can call itself. A local or global function prototype is declared as in ANSI C.

An example of a valid function declaration and definition is the following:

```
/* Declaration of the function prototype */

int factorial(int);

/* Definition of the function            */

int factorial(int n) {
  int i,res;

  res = 1;
  for(i=1 ; i <= n ; i +=1 ) res *= i;
  return(res);
}
```

The result of a function is given by the `return` instruction.

2.2.6 Cell type definition

A cell type definition is composed of a heading indicating its name, a set of variables and channels, and a body including the rules of the cell, as outlined below:

```
typecell cell_name(cell_parameters)
{
   channel definitions
   local variables definitions
   local functions definitions
   set of <condition> ==> <action> rules
}
```

A cell type definition begins by the keyword `typecell`. The cell type must be given a name and can have some parameters (like a function). The parameters also have to be scalar.

The special `main` cell type can have some parameters. All the parameters are set to their null element on the first creation of a `main` cell, when the program is run. Any further creation of more `main` cells will take the parameter values into account. These `main` cell parameters are not the command line parameters, which are described in 2.2.10.

Here are some examples of valid cell types:

- `typecell main() {...}` — declares the `main` cell type, without any parameters.
- `typecell object(symbol name) {...}` — declares the `object` cell type, with the symbolic parameter `name`.
- `typecell link(int i,symbol s) {...}`

The I/O channels are described under the definition headings, followed by the cell variables (see section 2.2.4) and the set of rules.

Rule declaration

A rule follows the syntax `[priority] condition ==> action`. The `priority` is optional. The default priority, if none is given, is 1. The `condition` part of the rule can be any expression returning a boolean. Function calls are allowed in a `condition`. Thus, cell variables and output channels can be modified only by evaluating a rule condition, even if the rule is not selected. The `action` part is a single instruction or a set of instructions between { }.

There are two particular `condition` parts for the rules: the `initially` and `finally` conditions. Either or both of the conditions can be present. There must be no more than one `initially` or `finally` rule. It is usual (though not necessary) to write the `initially` rule as the first one in the cell body, and the `finally` rule as the last one. For the use of these particular rules, see section 2.3.1.

A rule can have a set of subrules, called a *sequence*, and declared by `-->` or `-(i)->`, with `i` an integer. A sequence is used to specify a sequence of actions to be executed. For more information about subrules, see section 2.4.1.

Valid cell type declaration and definition are given below:

```
/* Declaration of the cell type prototype */

typecell object(symbol) {
  in int i;
  out symbol s;
}
```

```
/* Definition of the cell type            */

typecell object(symbol name) {

  in int i;
  out symbol s;
  boolean test;

  (i == 1) ==> test = TRUE;
  (i == 2) ==> {
     s = 'two';
     test = FALSE;
  }
}
```

This cell, named `object`, takes one symbolic parameter `name`, has one integer input channel `i`, one symbolic output channel `s`, one local boolean variable `test`, and two rules.

2.2.7 Variable, function and cell scopes

This section describes the scopes of the variables, the parameters, the channels, the functions and the cell types.

Variable scope

There are no global variables in ParCeL-1. But there are cell and function parameters, cell and function variables, and channels.

Cell parameters, variables and channels are visible from the entire cell, i.e. from the cell rules and from the cell local functions. A cell local function can manipulate the variables and the channels of its cell, but a global function cannot. The channels are the only variables visible from the other cells, and only after connection requests. Even if two cells share the same type, they cannot share their variables.

Function parameters and variables are visible only in the function body. If a local function variable or parameter has the same name as a cell parameter, variable or channel, then the local one hides the global one.

Function scope

Global functions are visible from everywhere, and cell local functions are only visible inside cells in which they are defined. Inside these cells, a local function can be called from the cell rules, and from the set of local functions. If a local function has the same name as a global function, and if it is declared before the function call, then the local function hides the global one. If the local function is not yet declared at the function call, then the global function will be called.

Cell scope

A cell type is visible from everywhere. Two different cell types cannot share the same name.

Name space

ParCeL-1 uses a single name space: a single name can denote a cell, a function and a variable at the same time. There will be no conflict, because the ParCeL-1 syntax is unambiguous.

2.2.8 Prototype declarations

A variable, a parameter, a channel, a function or a cell type can be referred to only after it has been defined or declared. If the definition follows the use, or is in another file, then a declaration before the use is necessary. An item can be declared and never defined, if it is not used.

Prototype syntax

Variables, parameters and channels can be defined only at the beginning of the cell type and the function definitions, before any instruction. Thus they are always defined at the beginning of their scope, and before any use.

On the contrary, functions and cell types can be defined after the beginning of their scope. In this case it is necessary to declare the prototype of the function or cell type. As in ANSI C, a function prototype is the type, the name and the parameter type list of the function, followed by a semicolon. For example the prototype declaration of the math function `pow()` is the following:

```
double pow(double,double);
```

A cell type prototype is more complex, since it has to include the parameter type list, but also the type and name of all the channels of the cell. For example, here are the prototype and the definition of a cell type:

```
/* Cell type prototype declaration.  */

typecell actor(int,double) {
  in  int    ch_in;
  out symbol chout[2];
}
```

```
/* Cell type definition.              */

typecell actor(int i, double x) {

  in  int    ch_in;
  out symbol chout[2];

  int a,b,c;

  ... ==> ....
}
```

Prototype localization

Global function and cell type prototypes can be declared anywhere in the set of the global function and cell type definitions. They are usually declared in header files (`.h`), which can be included in any source file (`.p`) which uses the corresponding functions or cell types.

Local function prototypes have to be declared in their cell type body, in the local function area, i.e. after local variable declarations and before cell rule declarations.

2.2.9 Control instructions

The usual control instructions like `if, for, while, do...while` are available and follow the same syntax in ParCeL-1 as in C. See the reference manual for more information about each of these instructions.

2.2.10 Input/output instructions

There are two kinds of input/output instructions in ParCeL-1 handled via libraries. Those of the std and hcifunc libraries and those of the hcicell and diskcell libraries. In the first case I/O are handled as instructions. In the second case, I/O are handled as requests and are executed via some dedicated I/O cells.

The complete documentation for each of these I/O libraries is given in the libraries reference manual; we only give here a brief outline of the various I/O libraries.

The std library

A ParCeL-1 program can get the command line parameters using two functions of the std library: `argc()` and `argv()`. The `argc()` function returns the number of the command line parameters including the name of the program. The function call `argv(n)` returns a symbol that is the n^{th} command line parameter. The first parameter, the program name, is returned by `argv(0)` and the last one is returned by `argv(argc()-1)`.

The hcifunc and diskfunc libraries

Using the hcifunc library, ParCeL-1 can only print and read its scalar types: `int, double, symbol, boolean`. The corresponding output functions are `printi, printd, prints, printb`. Thus, instructions like `printi(i), printi(2*3); printd(4.23); prints('symbol')` are valid.

The corresponding input functions are `geti, getd, gets` (a boolean cannot be read directly from the standard input). Each of these functions returns a result of a particular type. For example, if the variable `i` is integer, the instruction `i = geti()` will place into `i` the result of a keyboard input.

The use of the diskfunc library is very similar: this library provides funcitons to open and close files (`fopen` and `fclose`) and the I/O functions corresponding to those of the hcifunc library: `fprinti, fprintd, fprints, fprintb` and `fgeti, fgetd, fgets`.

These I/O libraries are easy to use but is not available on certain architectures (for example, on T-Node with Inmos C 2.01.10 compiler, see A.4 for details). These libraries are further described in the librairies reference manual.

The hcicell and diskcell libraries

These libraries provide the user with some pre-defined cells specialized in I/O operations. These cells have some input and some output channels, used to receive what is be output, and to send what has been input. Any cell of a ParCeL-1 program can create and connect itself to some I/O cells, and send I/O commands to it.

The hcicell and diskcell libraries are less easy to use than the hcifunc and diskfunc libraries but are available on every architecture.

2.2.11 Cell network management requests

A running ParCeL-1 program consists of a network of communicating cells: see figure 2.2. The cells are connected to one another by some channels. The operations allowed on this network can be divided into two main classes: the cell operations, and the channel operations.

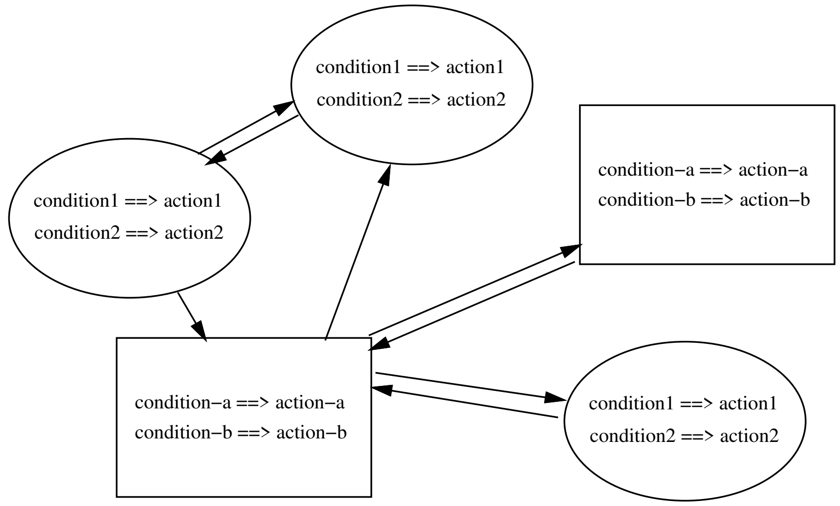

Figure 2.2: A network of communicating cells

Cell operations

ParCeL-1 allows the user to create, kill, awake, and sleep cells. The associated instructions are `create, kill, awake` and `sleep`.

The `create` request takes a cell type and returns a cell immatriculation. This immatriculation can be stored in a variable for further reuse, e.g. to connect the new created cell with another one, (see below). The newly created cell will be alive and will execute its first rule (the `initially` rule if it exists) at the next cycle. Example of use:

```
immat object new_cell;
...
new_cell = create object('elephant');
```

The `kill` request operates on a cell immatriculation and removes this cell from the network. The death of the cell will be effective only at a future cycle, depending on the complexity of the cell set of rules (see section `kill` in the reference manual for complete documentation). The communication channels connected to this cell remain available for read access and keep their last assigned values. Example of use: if `cell` has the type `immat`: `kill cell`.

The `awake` and `sleep` requests also operate on a cell immatriculation and stop or restart the running of the corresponding cell. A sleeping cell does nothing, i.e. a rule is neither evaluated nor selected. The output values of an asleep cell do not change. They remain equal to the values of the last cycle. A sleeping cell can be awakened by the `awake` request. Any cell can be put to sleep or awakened at any time. Since an asleep cell is inactive, it has to be awakened by another cell. Example of use:

```
immat object cell;
...
awake cell;
sleep self;
```

Channel operations

ParCeL-1 can create and destroy a channel, via the `connect` and `disconnect` requests. A channel can have any scalar type. The `connect` instruction connects an input channel of a given cell to an output channel of another one: in some way, the input channel is plugged in the output channel. The channels must be declared in the cell types. The channels to be connected with each other must be of the same type. The connection will be effective only at the next cycle. The complete syntax of these requests is given in the reference manual. Here is an example of a connection using the `connect` request. First we declare two cell types:

```
typecell foo() {
  in symbol s_foo_out;

  ...
}
```

```
typecell bar() {
  out symbol s_bar_in;

  ...
}
```

Before any connection is created, the input and output channels exist but are independent from each other: the channels are said to be *not connected*. When such a channel is read, the returned value is the zero of its type. The state of the cells is shown on figure 2.3.

Now, the following request is executed by the cell `other`, which knows the immatriculations of a `foo` and a `bar` cell. These immatriculations are called `foo_immat` and `bar_immat`.

```
typecell other() {
  immat foo foo_immat;
  immat bar bar_immat;

  ==> connect s_bar_in of bar_immat
          to s_foo_out of foo_immat;

  ...
}
```

Figure 2.3: Cells before any connection

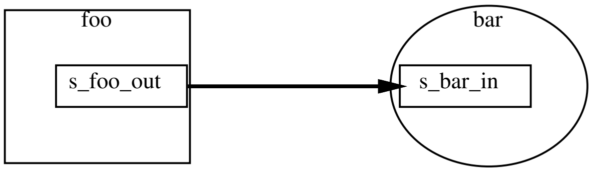

Figure 2.4: Cells after one connection

After execution of this `connect` request, `s_foo_out` and `s_bar_in` are not independent anymore. Any value taken by `s_foo_out` (by a mere initialization, as in `s_foo_out = 2`) will be also taken by `s_bar_in`, *at the next cycle*. The state of the two cells, after execution of the `connect` request, is shown on figure 2.4.

An output channel can be connected to several input channels, simply by executing a series of `connect` requests. A possible communication graph is shown on figure 2.5.

It is possible to write in an unconnected output channel. The data written is then stored in this output channel and will be available to some input channel(s) that may later be connected to it.

A pair of connected channels can be disconnected from one another using the `disconnect` request, which has the opposite effect of the `connect` request. A disconnected channel can be reconnected later. A disconnected output channel keeps its value, which will be available to some input channel(s) that may later be connected to it.

Since a channel is not always connected, it is possible to save memory by disposing of an unconnected output channel. This is done by using the `free()` function (see reference manual). The space used by this output channel is then released, and its value is erased. However, this channel can be used again for further connection.

In addition, it is possible to know whether a channel is connected or not, using the `connected` function. Applied to a channel name, it will return a boolean indicating if this channel is connected to another one or not.

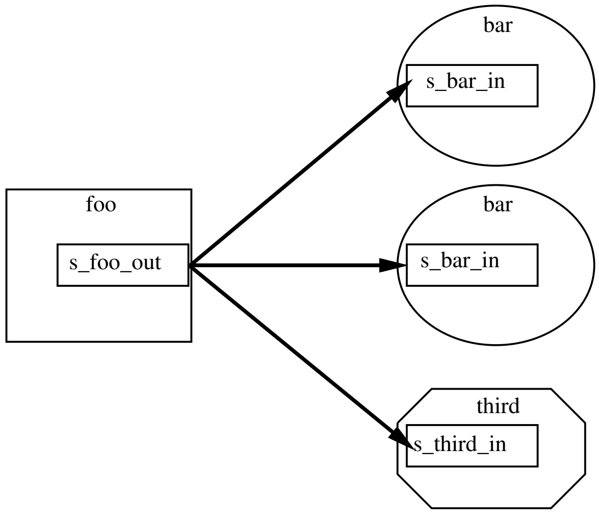

Figure 2.5: Communication graph between several cells

2.3 Program execution

2.3.1 Rule selection

When a `ParCeL-1` program is run, one cell of the `main` cell type is created. This cell will be in charge of creating others. During the execution phase, one or zero rule is selected in each running cell at each cycle. The selection of a rule follows the following guidelines:

- If the condition part of a rule is `initially` and it is the first cycle run by the cell, then this rule will be selected.

- If the condition part of a rule is `finally` and it is the last cycle run by the cell, then this rule will be selected.

- Otherwise, the condition part of each rule is evaluated. An empty condition is true.

- A rule can be selected only if its condition part is verified (true).

- If several rules can be selected, then the selected rule will be the one with the highest priority or the first selectable rule in case of equal priorities.

- If no rule has its condition verified, then no rule is selected.

2.3.2 End of execution

The execution of a program will stop only when the `halt` request has been called. Since `halt` is a request rather than an instruction, the current cycle will be fulfilled, and *all the cells* will be updated. The `halt` request can be executed at any time *by any cell.*

2.4 Advanced programming

2.4.1 Sequences of subrules

To execute a series of actions at different cycles, ParCeL-1 provides the notion of sequences of subrules. Using subrules, the programmer can specify the behavior of a cell for several further cycles, and specify explicitly the order of the actions to be executed.

A sequence of subrules is declared inside a rule (or main rule), using the operators `-->` or `-(i)->` with `i` as an integer. Here is a definition of a rule with subrules:

```
(i == 1)  ==>    output_channel = 1;
          -(3)-> output_channel = 3;
(i == 2)  -->    output_channel = 4;
```

This rule is entered when `(i == 1)` is true. At the first cycle `output_channel = 1` is executed. Then, 3 cycles later, `output_channel = 3` is executed. Then, one cycle later, if `(i == 2)` is true, `output_channel = 4` is executed. Actually `-->` is a short for `-(1)->`. See the Reference Manual for a complete description of the subrule mechanism.

2.4.2 Hints for efficient programming

Since a ParCeL-1 program can be run on multi-processor architectures, some supplementary programming rules must be considered to improve performance. These hints are only indicative, and should be taken as advice to write better and more efficient programs.

1. Avoid creating, connecting, killing, ... too many cells in one single cycle, so as not to cause an overflow in the request storage tables.

2. Avoid creating too many little cells, each doing a tiny job, or too few cells, each doing a huge job. Try to have a medium granularity. An overfine granularity may cause communication over-head, and too coarse a granularity may harm the load balance on the processors.

3. Avoid creating cells with short lifetimes, because cell creation is a time-consuming process.

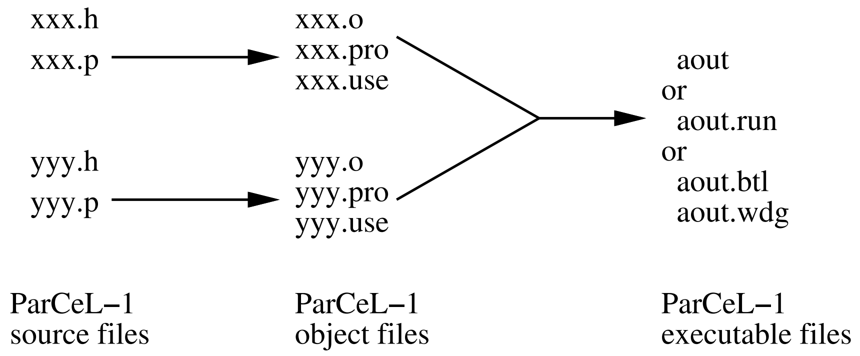

Figure 2.6: Example of a multi-file ParCeL-1 compilation

4. Minimize the number of channels between cells because channel management is a time-consuming process.

5. Of course, make at least as many active cells as the number of processors of the target machine in order to ensure efficient use of each processor.

2.5 The ParCeL-1 compiler

The compiler behaves for the most part like a C compiler. The compiler is called by the instruction `parcel`. Executable code will be produced only if the source code is free of errors; otherwise compilation error messages indicating the files and the lines where the errors appear in the source code will be displayed. The error messages are listed in appendix B.

2.5.1 ParCeL-1 compilation mechanisms

A ParCeL-1 compilation uses and generates several types of files (see figure 2.6):

- source files, suffixed `.p`,
- header files, usually suffixed `.h`,
- object files, suffixed `.o`,
- private files, suffixed `.pro` and `.use`, used during the link step with the object files,
- executable files, whose suffix depends on the target architecture and the C compiler called (e.g. `.btl` for T-Node MIMD computer),

- sometimes processor network configuration files, such as `.wdg` suffixed files for T-Node MIMD computer.

The compilation capabilities of ParCeL-1 are described below:

Command line parsing

The command line is completely parsed before compilation begins. Thus the order of the arguments (options and file names) is not important.

Executable file name

Default executable file name is `aout`, and its suffix is imposed by the target architecture (`aout.btl` for T-Node computer). The `-o` ParCeL-1 option allows the user to specify another executable file name, but its suffix is always imposed by the target architecture.

Directories of input and output files

By default input files are read and output files are written to the current directory. But input and output file names can be preceded by different access paths. In any case, `.o`, `.pro` and `.use` associated files are always read or written in a unique directory. For example:

```
parcel ../src/foo.p ../res/foo
```

will compile the `../src/foo.p` source file in the `../res/foo` executable file.

Compilation of a file list

ParCeL-1 can compile and link several files at a time by taking a file name list as a parameter. For example:

```
parcel xxx.p yyy.p zzz.p -o result
```

will produce the single `result` executable file, or `result.btl` if compiled for T-Node target architecture.

Compilation to object files only

The `-c` ParCeL-1 option allows the user to stop the compilation before the link step, and to generate only the `.o`, `.pro` and `.use` files. If this `-c` option is not used, the `.o`, `.pro` and `.use` files generated from each source file, will not remain after the end of the compilation.

Usually, a `xxx.p` source file is compiled into a `xxx.o` object file and `xxx.use` and `xxx.pro` private files. But with the `-c` and `-o` ParCeL-1 options it is possible to rename these files. In this case only the unsuffixed name (possibly with a path) should be given. For example:

```
parcel -c xxx.p -o object/foo
```

will produce `object/foo.o`, `object/foo.pro` and `object/foo.use` files. If the `-c` and `-o` options are used at the same time with a list of source files, they will all be compiled into the same object file, and the object files will be erased.

Compilation of object files

ParCeL-1 can link (`.o`) object files with other files, provided that each `.o` file is associated with its `.pro` and `.use` files. Private files `.pro` and `.use` can be erased only if the corresponding object file is not used and has to be erased too.

Library usage

The ParCeL-1 libraries to be included have to be specified in the command line with the `-l` option, without a space before the library name. It is necessary to include the header file of the library in the source file that uses it. Example:

```
/* File xxx.p                */

#include <math.h>
....


/* Compilation command line. */

parcel xxx.p -lmath
```

Specification of target architecture

The compilation target architecture can be specified with the `-target` option. It must be specified as well for the compilation step of sources to objects as for the linking step. All the object files used must have been generated for the same target. See section 2.5.3 for further explanation about the `-target` option.

Examples

Below are three possibilities for the compilation described in figure 2.6.

```
/* Complete compilation.                */

parcel xxx.p yyy.p zzz.p
```

```
/* Separate compilation of each file,     */
/* and further linking.                    */

parcel -c xxx.p
parcel -c yyy.p
parcel -c zzz.p
parcel xxx.o yyy.o zzz.o


/* Separate compilation of only one file. */

parcel -c xxx.p
parcel xxx.o yyy.p zzz.p
```

2.5.2 The preprocessor

ParCeL-1 source files can include some CPP commands. CPP is the preprocessor used by C. Useful commands are `#define` to define a macro, and `#include` to include a file. Included files surrounded by `<>` will be searched for in the system libraries, and included files surrounded by `" "` will be searched for in the current directory. Constants and macro-commands can also be defined using the CPP syntax.

2.5.3 Call of the compiler

The ParCeL-1 compiler handles the following switches (this message is produced by the instruction `parcel -help`):

```
-c:             Produce '.o' files, do not run linker.
-E:             Run source through the preprocessor
                (default output to stdout).
-g:             Generate code with runtime error checking
                (only on SISD architectures).
-s:             Do not warn symbol truncature.
-o file:        Set name of output file.
-lX:            Link with object library X.
-h n:           Set halt level to n (1-5).
-w n:           Set warning level to n (1-5).
-target X:      Generate code for C-compiler X
                (cc_sun4,gcc_sun4,cc_alpha,gcc_alpha)
                (cc_i386-linux,gcc_i386-linux,cc_hp9000s700,gcc_hp9000s700)
                (parc_1t,parc_nt,icc_1t,icc_nt,c89_t3d_1p,c89_t3d_np).
-cell n:        Set maximum cell registration number of all the CAM (>0),
```

```
                 default: 1000,
                 at least 1 per processor.
-channel n:      Set maximum channel number of all the CAM  (>0),
                 default: 2 x cell registration number.
-request n:      Set size of each request table of one processor to another (>0),
                 default: 1024 bytes.
-stack n:        Set ParCeL-1 stack size (in bytes),
                 default: 8192 bytes.
-prof:           Generate code for usage with C-profiler.
-timing:         Generate code producing timing information.
-help:           Print this message.
-ver:            Print ParCeL version number.
Suffix:          'p': ParCeL-1 source file.
Suffix:          'o': object file.
```

Complete description of the switches:

`-c` Suppress linking and produce a .o file for each source file.

`-E` Stop after the preprocessing stage; do not run the compiler itself. The output is preprocessed source code, which is by default sent to the standard output.

`-g` This option is only available on sequential architectures (see A.4 for details). If a program is compiled with the `-g` flag, a set of consistency checks will be done: division by zero, range of the index in the area accesses, consistency of the cell immatriculations. On error, the program will stop and an appropriate error message will be displayed indicating where the problem occurred.

`-s` (silent) Symbols in ParCeL-1 are limited to eight characters. If a symbol in the program is longer than eight characters, such as the symbol `'character'`, it will be truncated and a warning message will be displayed during the compilation, unless the `-s` flag is set.

`-o outputfile` Name the output file `outputfile`. The output file cannot be the same as the source file or any object file (the compiler will not overwrite these files).

`-l library` Link with the object library `library`.

`-w n` **and** `-h n` , the value of `n` being from 1 to 5: Set the warning and halting levels of ParCeL-1 run-time errors. Level one errors are the weakest errors, such as awakening an awake cell, and level five errors are the most important ones (see appendix B for the list of possible errors). The `-w` and `-h` options are used to set the level from which the errors will be reported or the program will be halted. The default level for these two options is 2, meaning that the errors whose level is greater or equal to 2 will be reported and will halt the program. Setting `-w` to 1 and `-h` to 5 will report every error but will make the program stop only for the most important ones. There is no use having a higher `-w` value than `-h` since all the reported errors will cause the halting of the program.

`-target` The compiler can produce executable code for several architectures. The target architecture is given by the `-target` option. If no explicit target is given, the default target specified during ParCeL-1 installation in the `CONFIG.H` file is chosen. The following table lists the different possible targets:

`-target` option	C compiler	Type of computer	Type of code
`cc_sun4`	cc	Sun 4	sequential
`gcc_sun4`	gcc	Sun 4	sequential
`cc_hp9000s700`	cc	HP 9000 S 700	sequential
`gcc_hp9000s700`	gcc	HP 9000 S 700	sequential
`cc_i386-linux`	cc	Linux PCs	sequential
`gcc_i386-linux`	gcc	Linux PCs	sequential
`cc_alpha`	cc	Alpha	sequential
`gcc_alpha`	gcc	Alpha	sequential
`parc_1t`	parc	Parsytec mtmsun boards	sequential
`parc_nt`	parc	Parsytec mtmsun boards	parallel
`icc_1t`	icc	T-node	sequential
`icc_nt`	icc	T-node	parallel
`c89_t3d_1p`	c89	Cray T3D	sequential (obsolescent)
`c89_t3d_np`	c89	Cray T3D	parallel

`-cell n` Set the total number of cell immatriculations that can be handled by ParCeL-1, i.e. the maximum number of cells that can exist at the same time on the set of processors. The default value is 1000, a suitable value should be at least 100 per processor. When an RF5 error appears (Table of cell immatriculations full) the program should be recompiled with an increased value for `-cell`.

`-channel n` Set the total number of channels that can be handled by ParCeL-1 on the set of processors. The default value is the double of the cell immatriculations number: the average number of channels per cell is 2. A suitable value should be at least 200 per processor. When an RF4 error appears (Tables of channels full) the program should be recompiled with an increased value for `-channel`.

`-request n` Set the size (in bytes) of the request table of one processor to another. The default value is 1024 bytes. When an RF3 error appears (Tables of requests full) the program should be recompiled with an increased value for `-request`.

`-stack n` Set the size of ParCeL-1's internal stack (in bytes). The default value is 8192 bytes. When an RF6 error appears (Full stack) the program should be recompiled with an increased `-stack` value.

`-prof` When the `-prof` option is specified, the executable code generated by ParCeL-1 can be used by a profiler to measure the time spent in each function of the users's program. The profiler to use is the one associated to the target C compiler. For example, with the target `gcc_sun4` the profiler to use is `gprof`; with the target `cc_sun4`, the profiler to use is `prof`. Refer to the documentation of your profiler for further details. This option is not available on every architecture (see A.4 for details).

`-timing` When the `-timing` option is specified, the execution of a program prints a summary of the configuration of the running architecture, and information about the time spent in the main phases of the VCM-1 cycles. Time is expressed in seconds. This option is not available on every architecture (see A.4 for details). An example of output follows:

```
#
# ParCeL-1 (1.3.2-beta) - c89_t3d_np - 32 PE.
#
# ( aout 100 1000 )       <command line>
#
#  32 PEs (  8 x  2 x  2) <network topology>

... (program output)

# Timing:
# - Local processing               : 4.563 s
# - Waiting after local processing  : 0.152 s
# - Request management             : 2.421 s
# - Waiting after request management: 0.039 s
# - Garbage collection             : 0.043 s
# - Data communication             : 9.774 s
# - Waiting after data communication: 0.028 s
# - Total execution time           : 17.021 s
```

`-help` Print the message above.

`-version` Show ParCeL-1 version number.

2.5.4 Using make with ParCeL-1

It is possible to write a `makefile` and to use the Unix `make` tool for a ParCeL-1 compilation, by including a special configuration file in the `makefile`. The reader is assumed to be familiar with the makefile notations. The directory `/..../parcel/config/make` contains this special configuration file (`default.mk`) and a makefile example (`make.example`). The `default.mk` file has to be included at the beginning of the user's `makefile`, as the example of makefile. The `default.mk` file defines:

- a macro which is the ParCeL-1 compiler name: `PCLC`;
- a macro to compile ParCeL-1 source files into object files: `COMPILE.pcl`;
- a macro to link ParCeL-1 object files into an executable file: `LINK.pcl`;
- a `CFLAGS` macro to specify the compilation flags (`-g`, `-h`, `-w`). This macro is used by the `COMPILE.pcl` and `LINK.pcl` macros;
- a `TARGET_ARCH` macro to specify the target architecture (`gcc_sun4`, ...), and which is used by the `COMPILE.pcl` and `LINK.pcl` macros;
- a `DEPENDFLAGS` macro, which can be used to write a `makedepend` rule, as shown in the `make.example` file;
- an implicit rule to create object files (`.o`) from ParCeL-1 source files (`.p`).

Thus, some makefile rules can be written to perform:

- explicit and systematic compilations, with the `PCLC` and `CFLAGS` and `TARGET_ARCH` macros (see rule `systematic` of `make.example`);
- implicit and economic compilations, with the implicit rule compiling source files into object files and with a rule using the `LINK.pcl` macro (see rule `economic` of `make.example`).

Before making economic compilations (that is, re-compiling only the necessary files) it is suitable to execute a `makedepend`, and to write a `depend` rule as in the `make.example` file, using the `DEPENDFLAGS` macro.

2.6 Parallel hardware support and configuration

The current ParCeL-1 compiler can produce code for multiprocessor computers: Parsytec mtmsun boards with the Par.C system 1.32, T3D computers, T-Node computers with the INMOS C compiler version 2.01.10. These are transputer based architectures, featuring MIMD processing, distributed

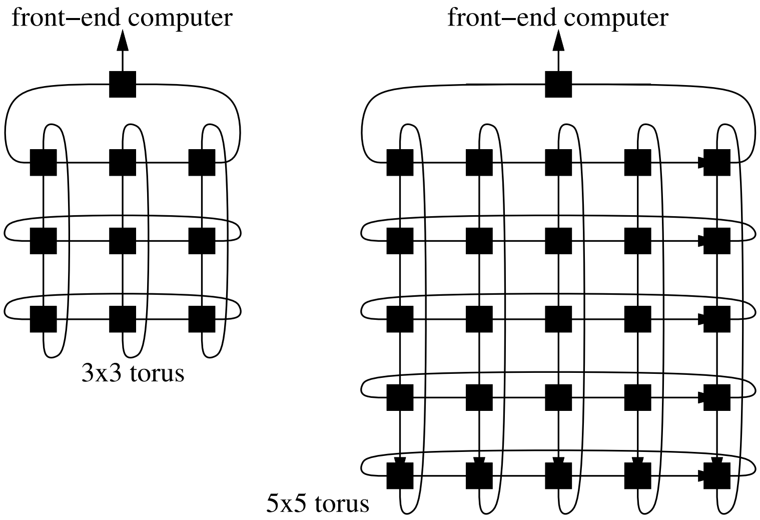

Figure 2.7: Examples of transputer torus topologies, with connection to a front-end computer.

memory, scalability. Currently, only one network topology is supported by ParCeL-1: the bidimensional square torus with an odd number of processors on its sides, with a connection to a front-end computer (see figure 2.7 for examples). Any square torus of odd size is allowed (3x3, 5x5, 7x7, ...), but ParCeL-1 has currently been tested only on a 3x3 torus.

Thus, to install ParCeL-1 on a transputer-based computer, it is necessary to create an odd size square torus topology before running a program. Network topology configuration is an architecture dependant operation, thus we cannot present a general approach. We present here the three current ParCeL-1 implementations. The reader is assumed to be familiar with the three environments (Parsytec, T-Node and Cray).

2.6.1 Using a Parsytec mtmsun board

The configuration of the transputer network is described in a file which is compiled with other ones into the `ulti` program. Running this program creates the desired hardware configuration on the transputer board until another configuration is created. This hardware configuration is independant of the installation of ParCeL-1. An example of configuration file of the transputer network required for ParCeL-1 is provided with the ParCeL-1 environment:

```
/.../parcel/config/parc_nt/mtmsun.ncu
```

The compilation of a `ParCeL-1` source produces a `.run` file, which is the transputer executable file. Before running it (or any program) the transputer network must be reset with the `makepath` and `reset` tools of the Parsytec environment. After this step, the `xxx.run` program can be executed by `run xxx`.

2.6.2 Using a T-Node with one processor

The `ParCeL-1` compilation for `icc_1t` with a T-Node computer produces a `.wdg` and a `.btl` file. The first one is a hardware configuration file, and the second one is the executable file. After the `ParCeL-1` compilation, it is necessary to:

- reset the T-Node with the `irsub` tool of the INMOS-C environment;
- configure the transputer network with the `itsw` tool of the INMOS-C environment and the `.wdg` file: `itsw xxx.wdg`;
- execute the program with the `irun` tool of the INMOS-C environment and the `.btl` file: `irun xxx.btl`.

2.6.3 Using a T-Node with several processors

With a T-Node computer, it is necessary to specify in a single file the hardware topology and the destination of the tasks constituting the program. This file is used during any INMOS-C compilation, and will be used during each `ParCeL-1` compilation with the `icc_nt` target. The `ParCeL-1` environment provides an example of this configuration file:

```
/.../parcel/config/icc_nt/vcm.icc_nt.cfs
```

To adapt this example to a specific topology, copy this file in the current directory of the `ParCeL-1` compilation and change the value of the side size variable. This is the `Side` variable in the top of the file. By default the `Side` variable value is 3. It is necessary to have a copy of this file in the compilation directory to make a `ParCeL-1` compilation with the target `icc_nt`.

The `ParCeL-1` compilation uses this configuration file, and produces a `.wdg` and a `.btl` file. The first is a hardware configuration file, and the second is the executable file. After the `ParCeL-1` compilation, it is necessary to:

- reset the T-Node with the `irsub` tool of the INMOS-C environment,
- configure the transputer network with the `itsw` tool of the INMOS-C environment and the `.wdg` file: `itsw xxx.wdg`,
- execute the program with the `irun` tool of the INMOS-C environment and the `.btl` file: `irun xxx.btl`.

2.6.4 Using a Cray T3D

The executable code produced on a Cray Y/MP computer for a Cray T3D massively parallel computer can be run in two ways:

- Interactively: by typing `<prog_name> -npes <number>` to precise the number of processors to be used. On some systems, the maximum number of usable processors may be less than the number of processors of the T3D system.
- In batch mode: by submitting the job to the Cray queuing system NQS, using the command `qsub <script>`. In this case, all the processors of the T3D can be used.

Chapter 3

ParCeL-1 reference manual

In this chapter, a complete description of each ParCeL-1 operator and keyword can be found. Some examples of use are given in each description.

`+ - * /`

nature: common arithmetic operators

description: These operators take integer or floating point operands. The result is of type integer if all operands are of type integer. If at least one operand is of type `double` then the result is of type `double`. The priorities are the usual ones for mathematical operators. The notation used is infix.

As in C, the division of two integers will be integer. To enforce floating point division, the `ddiv` operator has to be used. To enforce integer division of floating point operand(s), the `idiv` operator has to be used.

examples:

```
/* Priority example: two equivalent lines.        */

res = a * b + c / d - e;
res = (a * b) + (c / d) - e;
```

%

nature: common arithmetic operator

description: This operator executes a modulus operation. It applies only to integer operands and has an integer result.

example:

```
/* Simple example of modulo computing.             */

int a, b, c;

==> a = b % c;
```

> < >= <=

nature: common arithmetic comparison operators

description: These operators apply to two arithmetic operands, and produce a boolean result. Each of their operands can be of type integer or floating point.

examples:

```
/* Simple arithmetic comparison.                          */

if (a > 10) ....;

/* Computation of a boolean value.                        */

bool_var = a <= b;
```

== !=

nature: equality and inequality test operators

description: These operators take two operands of compatible types, and give a boolean result. It is possible to compare two integer operands, or an integer and a floating point operand, or two booleans, or two symbols, or two immatriculations of same cellular type.

examples:

```
/* Equality test of integer operands.              */

if (int_var == 12) .....

/* Comparison of an immatriculation to NULL.       */

if (foo_immat != NULL) ....
```

! && || (not, and, or)

nature: common boolean operators

description: These operators apply only to boolean operands, and give a boolean result. Their priorities are in this order: `! && ||`, with `!` the highest.

examples:

```
/* Priority example: two equivalent lines.        */

boolean a, b, c;

==> if (a && !b || c) ....
==> if ((a && (!b)) || c) ....
```

=

nature: assignment operator

description: This operator assigns scalar operands, and output channels. The input channels and the private variable `self` cannot be used as left operand of an assignment, since they are read-only variables, but they can be used as right operand.

example:

```
/* Output channel assignment.                          */

symbol_channel_out = 'foo';
symbol_channel_out = symbol_channel_in;
```

`+= -= *= /= ddiv= idiv= %=`

nature: arithmetic assignment operator

description: These operators perform a combined computation and assignment on their left operator. Their operands have to be of arithmetic types (`int` or `double`). For example `x+=a+b;` is the same as `x = x + (a+b);`, except that `x` is evaluated only once, not twice. This is important when writing a table access where the index evaluation uses a function call which can modify the cell variables. The left operand has to be writable, and can be an output channel, a common variable, or a writable private variable.

These operators produce arithmetic results, but the result of the computation and the left operand have to be of compatible types. For example, with `i` of integer type, `i += 1;` is valid, but `i += 1.0;` is not. In this last case the result is of floating point type and cannot be assigned to an integer variable. On the contrary `x += 1;` is valid with `x` of type `double`. Refer to the description of the corresponding arithmetic operator for details on the computation method and type constraints.

example:

```
/* Output channel computation and assignment.     */

int_channel_out *= 2;
int_channel_out *= 2 + i;

/* Floating point computation and assignment.     */

double_var /= 3.14159;
```

==>

nature: main rule specifier

syntax: `[[p]] [<expression>] ==> <instruction>`

description: `==>` specifies a main rule of a cellular type. A main rule has one necessary and two optional arguments:

- the rule priority: `[p]` (optional) is a positive floating point value between square brackets `[]`. If no priority is indicated the rule takes the default priority `1`. If two rules are given the same priority, the first declared will have priority.
- the rule condition: `<expression>` (optional) indicates whether the rule is executable. When a cell is activated, all the conditions of the rules are evaluated, and the rule to execute is chosen from the set of executable rules. A condition may be of any scalar type. If it is not a boolean expression, then the condition is an implicit inequality test of the expression to the 'zero' of its type. An empty condition is always true. If no condition rule is true, no rule is executed. If several rules are executable, the one with the highest priority is executed.
- the rule body: `<instruction>`, is the action executed when the rule is executed. It can be a simple instruction or a set of instructions between braces `{}`.

example:

```
/* Simple rule examples.                           */

           b ==> ...;              /* rule 1    */
[2.0]   a && b ==> ...;            /* rule 2    */

/* If only b is true, rule 1 is executable.        */
/* If a and b are true, rule 1 and 2 are           */
/* and rule 2 is executed.                         */
```

--> -(i)->

nature: subrule specifier

syntax: [<expression>] -(i)-> <instruction>

description: -(i)-> specifies a rule associated with a main rule (==>). A main rule can have an unlimited number of subrules. The declarations of these subrules immediatly follow the main rule declaration in the set of rules of the cell:

```
a  ==>   <action 1>;
b -(3)-> <action 2>;
c -(2)-> <action 3>;
```

The meaning of such a rule declaration is the following: if condition a is true at cycle n, <action 1> will be executed, and the cell will execute <action 2> at cycle $n+3$ with the only condition being that b is true at this cycle. The value of the condition a or the values of other main rules conditions will have no effect on the choice of the rule to execute at cycle $n+3$. During cycles $n+1$ and $n+2$ the cell will not execute any rule. This mechanism will go on with the next subrule: the cell will execute <action3> at cycle $n+5$ (two cycles after cycle 3) if the condition c is true at this cycle.

More precisely, when executing a main rule with subrules, the cell enters in a "sequence" of rules. It can leave this sequence and return to the choice of its complete set of main rules only in two cases. First, when the sequence is finished, i.e., when the last subrule has been executed. Second, when the condition of a subrule is false at the cycle it is evaluated. In this last case the sequence is broken, and no rule is executed at this cycle, but all main rules will be considered again at the next cycle. In the above example, if condition b is false at cycle $n+3$ then no rule is executed at this cycle, and all main rules conditions will be evaluated again at cycle $n+4$.

If a subrule condition is empty, then it is always true, as in the main rule case. The subrule specifier --> is the same as -(1)->, it is just a short notation. The special rules initially and finally can have subrules like any main rules.

examples:

```
/* Initial rule with subrules.                        */

initially  ==>  ....;
a || b     -->  ....;
          -(2)-> ....;
```

```
/* Main rules with their subrules.                  */
x ==> ....;
  --> ....;
  --> ....;

y ==> ....;

z ==> ....;
  --> ....;
```

awake

nature: cell awake request

syntax: `awake <cell_immatriculation>;`

description: This request awakens any cell of the cellular network, using its immatriculation. The target cell will be awake and active only at the next cycle. If the target cell is already awake, the `awake` request has no effect on the cell but generates a low level error.

example:

```
/* Awake of a cell from another.                    */

immat foo immat_foo;

....
awake immat_foo;
```

boolean

nature: scalar type specifier

description: `boolean` declares a boolean parameter, variable, channel or function. The two possible boolean values are `TRUE` and `FALSE`. The applicable operators for boolean variables are: `&&`, `||`, `==`, `!=` and `=`.

examples:

```
/* Boolean variable definition                        */

boolean i, j, tab[2][3];

/* Definition of a function returning a boolean result.*/

boolean f(...) {...}
```

connect connectiffree

nature: channel connection requests

syntax:

```
connect <input channel> of <immatriculation of reader cell>
     to <output channel> of <immatriculation of writer cell>;

connectiffree <input channel>  of <immatriculation of reader cell>
     to <output channel> of <immatriculation of writer cell>;
```

description: These requests connect an input channel to an output channel of any cell if their types are compatible. It is necessary to know the names of the two channels, and the immatriculations of the two cells. If a cell wants to request a connection for one of its own channels it has to use its self immatriculation (`self`).

The connection will be effective only at the beginning of the next cycle, but data written to output channels before the connection will not be lost. When a data is written to an output channel, it is stored in a buffer even if the output channel is not connected at that time. Later, when the connection is established, the input channel reads the last value stored in the buffer. Thus, if channels are connected and a piece of data is written to the output channel, this data will appear in the input channel at the next cycle. But if the connection of this channel is tested just after the `connect` instruction (in the same cycle) the result will be negative, unless the channel was used in an earlier connection. The `connected` test will be positive only at the next cycle, when the connection is effective.

An output channel may have several readers. An output channel can be connected to several input channels, simply by executing several connection requests. But an input channel cannot have more than one writer. If several `connect` requests are executed on the same input channel, only the last one will be effective. A `connect` request begins by destroying the previous connection if there is one. On the contrary, the `connectiffree` establishes a connection only if the input channel is free, i.e.: not already connected.

If connections are attempted for channels of dead cells, an error flag is positioned. If the error is greater or equal to the maximum authorised level (option `-h` of the ParCeL-1 compiler) the program will stop at the end of the current cycle, and an error message will be printed.

When a connection request is compiled, the prototypes of the used cellular types must be known. Thus, if the cellular types are not already declared, their prototypes must be declared.

example:

```
/* Connection, and sending of data.  */
out int a_out;
immat foo immat_foo;

...
connect a_in  of immat_foo
     to a_out of self;
a_out = 2;
...
```

connected

nature: boolean predefined function

prototype: `boolean connected(<channel_name>);`

description: This predefined function takes a channel as its argument, and tests whether it is connected. The channel can be of any type and can be an input or an output channel, but it has to belong to the cell which calls the `connected` function: a cell cannot test the state of a channel of another cell. Thus, this function can be called from a rule inside a cell, or from a cell local function, but not from a global function. The result is available immediately and illustrates the state of the channel in the current cycle. The result is of type boolean, `TRUE` if the channel is connected, and `FALSE` otherwise. An output channel is connected if it has at least one reader, and an input channel is connected if it has one writer (exactly).

examples:

```
/* Example of if...else test about a channel state. */

in int channel_in;
boolean boolean_var;

....
if (connected(channel_in))
  res = channel_in;


/* Storage of a channel state.                      */

....
boolean_var = connected(channel_in);
```

create

nature: cell create request

syntax: `create <cell_name>(<list of parameters of cell>);`

description: This request creates a new cell having the specified type (`<cell_name>`) and the specified parameters. This new cell is automatically created on the least loaded processor. The user does not need to specify a host processor for the new cell.

The `create` request returns the new cell immatriculation. This result can be stored in a variable of type immatriculation of the newly created cell. If the new cell could not be created, because of a lack of memory, then an emergency error signal will be activated, and the program will stop at the end of the current cycle with an error message.

examples:

```
 /* Simple create request.                          */

 create foo('bar',12);

 /* Create request with storage of the  new cell   */
 /* immatriculation.                                */

immat foo foo_immat;

....
foo_immat = create foo('bar',12);
```

ddiv

nature: common arithmetic operator

description: This division operator takes two operands. Each of these operands can be of type integer or floating point, and the result is always of type `double`. The result is the basic result of the floating point division of the operands. Floating point division has the same priority as multiplication.

examples:

```
/* Simple examples of floating point division.    */

double res;
int a, b;
double x, y;

==> {
  res = a ddiv b;
  res = x ddiv y;
  res = a ddiv y;
  res = x ddiv b;
}
```

disconnect

nature: channel disconnection request

syntax:

```
disconnect <input channel> of <immatriculation of reader cell>
     from <output channel> of <immatriculation of writer cell>;
```

description: This request disconnects an input channel from an output channel. It is necessary to know the names of the two channels, and the immatriculations of the two cells. If a cell wants to disconnect one of its channels, it has to use its self immatriculation (`self`).

The disconnection will be effective only at the beginning of the next cycle. The data written to the output channel during the same cycle as the disconnection request will never arrive at the input channel. The disconnection will be effective before. But during the cycle of execution of the disconnection request, the channels are still connected, and data can pass through. If the connection of a channel is tested (`connected`) just after a `disconnect` request (in the same cycle) the result will be positive: the connection still exists. The `connected` test will be negative only at the next cycle, when the disconnection is effective, unless if it has been reconnected before the test or if it is an output channel still connected to some other input channels.

An attempt to disconnect channels which are not connected, or channels of dead cells will activate an error flag. If the corresponding error level is greater or equal to the maximum authorised error level, (option `-h` of the `ParCeL-1` compiler) the program will stop at the end of the current cycle, and an error message will occur.

When a disconnection request is compiled, the prototypes of the used cellular types must be known. Thus, if the cellular types are not already declared, their prototypes must be declared.

example:

```
/* Disconnection.  */

out int a_out;
immat foo immat_foo;

...
disconnect a_in  of immat_foo
      from a_out of self;
```

do

nature: loop statement

syntax: `do <instruction> while(<expression>);`

description: Executes the `<instruction>` as long as the condition `<expression>` is true. At each step the condition is evaluated after the execution of the `<instruction>`. The `<instruction>` will be executed at least once, even if the condition is false at the beginning. The halting condition given by `<expression>` can be of any scalar type. If it is not a boolean expression, then the condition is an implicit inequality test between the expression and the 'zero' of its type.

example:

```
/* Factorial computing using a 'do' statement.    */

res = 1;
i = 1;
do {
  res *= i;
  i += 1;
} while (i < 10);
```

double

nature: scalar type specifier

description: `double` declares a double precision parameter, variable, channel or function. In ParCeL-1, the `double` type is translated to the C double type (the C language being taken as a target language for ParCeL-1 compilation). Thus, the representational capabilities of the `double` type are the same as those of the underlying C compiler.

examples:

```
/* Definition of variables of type double.         */

double i, j, tab[2][3];

/* Definition of a function returning a double.    */

double f(...) {...}
```

FALSE

nature: boolean constant.

description: Booleans have two possible values: `TRUE` and `FALSE`. These values are not integer values, and are different from `1` and `0`.

example:

```
/* Boolean variable initialized with FALSE.       */

bool_var = FALSE;
```

finally

nature: final rule specifier of cellular types

description: `finally` specifies the final rule of the cellular type declaration. The final rule of a given cell is activated only one time, during the first cycle following a `kill` request on the cell.

If the final rule contains a sequence (subrules `-->`), its execution can take more than one cycle. In fact `finally` can be interpreted as a rule condition, which would be true only during the first cycle following the request for the death of the cell. Lastly, the execution of the final rule is not affected by the awake or asleep state of the cell: even a sleeping cell will execute its final rule if it dies.

examples:

```
/* Cellular type with final rule executed in      */
/* one cycle only.                                */

typecell foo()
{
 ...     ==> ....;

 finally ==> ....;
}

/* Cellular type with finally rule executed in    */
/* one cycle minimum and three cycles maximum.    */

typecell bar()
{
 ...     ==> ....;

 finally ==> ....;
 ...     --> ....;
 ...     --> ....;
}
```

for

nature: loop statement

syntax:

```
for(<assignment>[,<assignment>]*;<expression>;<assignment>[,<assignment>]*)
  <instruction>
```

description: The first list of assignments is executed only one time, at the beginning of the `for` execution. Then, the `<instruction>` is executed as long as `<expression>` is true, and is followed at each step by the execution of the second list of assignments. The halting condition given by `<expression>` can be of any scalar type. If it is not a boolean expression, then the condition is an implicit inequality test between the expression and the 'zero' of its type. If the condition is false at the beginning, only the first list of assignments is executed, and the `for` exits without executing the `<instruction>`.

examples:

```
/* Simple function call in a for loop.             */

for(i = 0; i < 10; i += 1)
  bar(i);

/* More complex for loop.                          */

for(i = 0, res = 0; i < 10 && res < 1000; i += 1) {
  res += i;
  f(res);
}
```

free

nature: channel free function

syntax: `free(<output_channel_name>);`

description: This function erases an unconnected output channel of the cell, using the name of the channel. The output channel content is lost, and some memory space is recovered. The `free()` function is useful to optimize programs and minimize the communication time. It is nevertheless a dangerous function, because the content of the channel is lost. This function is only effective on unconnected output channels; otherwise it generates a low level error.

example:

```
/* Erasure of an output channel.                  */

out int out_channel;

....
free(out_channel);
```

halt

nature: complete stop request

syntax: `halt;`

description: This request stops the program. After a `halt`, all the cells of all the processors will be stopped at the end of the current cycle, and the program will finish. This is the only way to stop a ParCeL-1 program. This request can be called in a rule, a local function or a global function. The cell which executes this request is stopped only at the end of the current cycle, like all the other cells: specifically, if some instructions follow the halt request in the same rule, they will be executed.

example:

```
/* Example of stop of a program.                        */

... ==> halt;
```

idiv

nature: common arithmetic operator

description: This division operator takes two operands. Each of these operands may be of type integer or floating point, and the result is always of type integer. The result is the integer part of the floating point division of the operands. Integer division has the same priority as multiplication.

examples:

```
 /* Simple examples of integer division.          */

 int res;
 int a, b;
 double x, y;

 ==> {
   res = a idiv b;
   res = x idiv y;
   res = a idiv y;
   res = x idiv b;
}
```

if...else

nature: conditional action

syntax: `if (<expression>) <instruction_1> [else <instruction_2>]`

description: Evaluates the `expression` between `()`, and executes `instruction_1` only if `expression` is true. If the `else` statement is present, then `instruction_2` is executed when `expression` is false. The condition can be a boolean expression, or another scalar typed expression. If it is not a boolean expression, then the condition is an implicit inequality test between the expression and the 'zero' of its type. The integers are compared with `0`, the doubles with `0.0`, the symbols with `NIL` and the immatriculations with `NULL`.

examples:

```
/* 'if-else' statement with explicit boolean     */
/* condition.                                    */

if (bool_var1 || int_var < 10)
  ....;
else {
  ....;
  ....;
}

/* 'if' statement with implicit  zero comparison */
/* and its equivalent complete 'if' statement.   */

if (int_var + double_var)
  ....;

if (int_var + double_var != 0.0)
  ....;
```

immat

nature: scalar type specifier

syntax: immat <cell_type_name> <identifier_list>

description: Associated with a cell type name, `immat` defines a ParCeL-1 scalar type. The parameters, variables, channels and functions having this type will contain or return some cell immatriculations of the associated cell type. No immatriculation cast is possible: the immatriculations of two different cell types are considered as two different ParCeL-1 types. `immat` is not a type by itself, but must always be associated with a cell type name. The null immatriculation is `NULL`.

examples:

```
/* Definition of immatriculation variables of      */
/* cells of type foo.                              */

immat foo i, j, tab[2][3];

/* Definition of a function returning an           */
/* immatriculation of cells of type foo.           */

immat foo f(...) {...}
```

in

nature: input channel specifier

description: `in` specifies an input channel. It must be associated with a variable declaration in the channel declaration part of the cell type declaration. Input channels can have any ParCeL-1 type.

examples:

```
/* 3 input channels of type int.                    */

in int i, j, k;

/* Two dimensions table of input channels of        */
/* immatriculations of cell type 'foo'.             */

in immat foo tab[2][3];
```

initially

nature: initial rule specifier of cellular types

description: `initially` specifies the initial rule of the cellular type declaration. The initial rule is activated one time only, during the first cycle in which the cell exists. But if the initial rule contains a sequence (subrules `-->`), its execution can take more than one cycle. In fact `initially` can be interpreted as a rule condition which would be true only during the first cycle of the cell. Lastly the execution of the initial rule is not affected by the awake or asleep state of the cell: even a sleeping cell will execute its initial rule.

examples:

```
/* Cellular type with initial rule executed in    */
/* one cycle only.                                */

typecell foo()
{
 initially ==> ....;

 ...       ==> ....;
}

/* Cellular type with initial rule executed in    */
/* one cycle minimum and three cycles maximum.    */

typecell bar()
{
 initially ==> ....;
 ...       --> ....;
 ...       --> ....;

 ...       ==> ....;
}
```

int

nature: scalar type specifier

description: `int` declares an integer parameter, variable, channel or function. In ParCeL-1, the `int` type is translated to the C int type (the C language being taken as a target language for ParCeL-1 compilation). Thus, the representational capabilities of the `int` type are the same as those of the underlying C compiler.

examples:

```
/* integer variable declaration.                      */

int i, j, tab[2][3];

/* Definition of a function returning an integer.  */

int f(...) {...}
```

kill

nature: cell kill request

syntax: `kill <cell_immatriculation>;`

description: This request kills any cell of the cellular network. The only parameter is the immatriculation of the cell to be killed. If a cell wants to kill itself, it has to execute: `kill self;`. Like the other requests, the `kill` request does not take effect immediately. When the killing of a cell is executed, nothing changes at the current cycle, but at the next cycle the target cell enters a new state in which its death is planned. When the cell is activated, if it is executing a sequence of rules it continues without change until the sequence is done, and then begins to die. Otherwise it begins to die immediately.

When a cell begins to die, if it has a final sequence, i.e. a `finally` rule, it begins to execute it, and continues until this final sequence is over. If the cell has no final sequence or has finished it at the previous cycle, its input channels are disconnected, and the majority of its memory structure is released. The cell will not accept any new request, except `disconnect` requests. In particular, it is no longer possible to establish connections with it.

Each of the output channels of the dying cell will be erased only when all its readers are disconnected from it. The results of the output channels will persist for previously connected reader cells, until they have explicitly finished to use them. In this way it is possible to kill a cell without disconnecting it beforehand, and without disrupting the reader cells.

Finally, when all the output channels of the dying cell have been erased, its last memory structures are released, and the cell is really dead.

examples:

```
/* Kill of an other cell.                                 */

immat foo immat_foo;

....
kill immat_foo;

/* Suicide.                                               */

kill self;
```

NIL

nature: symbolic constant

description: The 'zero' of a variable of type `symbol` is `NIL`.

example:

```
/* Symbolic variable initialized with NIL.        */

symb_var = NIL;
```

NULL

nature: immatriculation constant

description: The 'zero' of an immatriculation variable (of type `immat`) is `NULL`, whatever the cell type of the immatriculation.

example:

```
/* Test on an immatriculation variable.          */

if (immat_var != NULL) {
...
}
```

out

nature: output channel specifier

description: `out` specifies an output channel. It has to be associated with a variable declaration in the channel declaration part of a cell type declaration. Output channels can have any ParCeL-1 type.

examples:

```
/* 3 output channels of type int.                        */

out int i, j, k;

/* Two dimensions table of output channels of          */
/* immatriculations of cell type 'foo'.                */

out immat foo tab[2][3];
```

return

nature: break and return from function call statement

description: `return` forces a function to stop and to return to the calling environment. If `return` is used without parameters then no value is returned, which is correct (and necessary) if the function is of type `void`; in case of non-void functions the returned value is random. If there is a parameter (`return(...);`) its type has to be compatible with the function type. If the function body has no `return` statement, and the function does not have the type `void`, then the returned value is random.

examples:

```
/* Example of return from void function.        */

void foo()
{
 ....
 if (...)
   return;
 ....
}

/* Example of return from typed function.        */

int bar()
{
 return(12);
}
```

self

nature: cell private variable

description: This private variable is read only, and allows cells access to their own immatriculation. This is necessary to allow a cell to make request on itself, or to communicate its immatriculation to other cells using channels or parameters.

examples:

```
/* Send of self immatriculation to another cell.  */

typecell foo()
{
 out immat foo out_ch;

 ==> out_ch = self;
}

/* Cell suicide.                                  */

 ==> kill self;
```

sleep

nature: cell sleep request

syntax: `sleep <cell_immatriculation>;`

description: This request allows a cell to sleep another one whose immatriculation is given as a parameter. If the target cell is executing a sequence, it is slept at the cycle following the end of the sequence, otherwise it is slept immediately at the cycle following the request. When it is asleep, a cell does not execute any rule until it is awakened. However, the `initially` and `finally` rules are not affected by the `sleep` request. These special rules are executable even if the cell is asleep. If the target cell is already asleep, a sleep request has no effect on it, but generates a low level error.

example:

```
/* A cell sleeping another one.                        */

immat foo immat_foo;

....
sleep immat_foo;
```

symbol

nature: scalar type specifier

description: `symbol` declares a symbolic variable, parameter, channel or function. Symbols are *words* which are 1 to 8 characters long. Symbols are written between quotes, as `'bird'`, `'1 to 5'` or `NIL`, which is the null symbol. The `symbol` type is scalar. The applicable operators are `==` and `!=`.

examples:

```
/* Symbolic variable declaration.                  */

symbol i, j, tab[2][3];

/* Definition of a function returning a symbol    */

symbol f(...) {...}
```

TRUE

nature: boolean constant.

description: Booleans have two possible values: `TRUE` or `FALSE`. These values are not integer values, and are different from `1` and `0`.

example:

```
/* Boolean variable initialized with TRUE.       */

bool_var = TRUE;
```

typecell

nature: cell type specifier

syntax: `typecell <cell_type_name>(...) {...}`

description: `typecell` declares a cell type or a cell type prototype. A cell type includes:

- some input and output channel declarations;
- some local variable declarations;
- some local function prototype declarations;
- some local function declarations;
- some `<condition> ==> <action>` rules.

A cellular prototype includes only the declarations of the input and output channels. Declaring a cellular prototype is not always necessary: the only rule is that the prototype of each cell type must be known when this cell type is used. For example, if `create foo()` is situated in a given rule, then the `foo` cell type or its prototype must have been previously declared.

Input and output channel declarations must come first, and rules must come after all the other declarations. Input and output channels can be declared in any order. The order of these declarations can change between the cell type the cell type prototype, but each channel must be present in both declarations.

examples:

```
/* Cellular prototype declaration.          */

typecell main(int, double) {
  in int res;    /* input - output channels.*/
  out double x;
}

/* Cell type declaration.                   */

typecell main(int i, double x) {
  out double x;   /* input - output channels.*/
  in int res;
  int a;         /* local variables.        */
  int f() {      /* local functions.        */
```

```
      ...
    }
    ==> ...;        /* rules.                    */
  }
```

void

nature: void type specifier

description: `void` is used only for functions returning no result. This specifier is used both in functions and function prototype declarations.

example:

```
/* Void function declaration.                    */

void foo_bar(...) {...}
```

while

nature: loop statement

syntax: while(<expression>) <instruction>

description: Executes the <instruction> as long as the condition <expression> is true. At each step the condition is evaluated before executing the <instruction>. If at the beginning the condition is false, then the while exits without executing the <instruction>. The halting condition given by <expression> can be of any scalar type. If it is not a boolean expression, then the condition is an implicit inequality test between the expression and the 'zero' of its type.

example:

```
/* Factorial computing using a 'while' statement. */

res = 1;
i = 1;
while (i < 10) {
  res *= i;
  i += 1;
}
```

Chapter 4

Libraries reference manual

In this chapter is a complete description of the standard libraries provided with ParCeL-1:

- hcifunc: keyboard and screen I/O functions.
- diskfunc: disk I/O functions.
- hcicell: keyboard and screen I/O cells.
- diskcell: disk I/O cells.
- math: mathematic functions.
- std: common useful functions.

4.1 hcifunc

The hcifunc (human-computer interaction functions) library provides the user with a set of simple I/O functions. This library is currently not available on every architecture (see A.4 for details).

The functions in the hcifunc library are accessible only if the file `hcifunc.h` is included in the source code, and if the code is compiled with the `-lhcifunc` option.

printb, printd, printi, prints, printq

nature: input/output functions

prototypes:

```
#include <hcifunc.h>

int printb(boolean);
int printi(int);
int printd(double);
int prints(symbol);
int printsq(symbol);
```

description: These functions print their unique argument on the standard output. The result is 1 if the output succeeded, or 0 if it did not.

- `printb` print a boolean.
- `printd` print a floating point number (double).
- `printi` print an integer.
- `prints` print a symbol.
- `printq` print a symbol between quotes ('').

examples:

```
/* Print a symbol.                          */

prints('hello');

/* Print an integer.                        */

printi(i);
```

printtab, printnl

nature: input/output functions

prototypes:

```
#include <hcifunc.h>

int printtab();
int printnl();
```

description: `printtab` prints a tabulation character; `printnl` prints a new-line character. The result is 1 if the output succeeded, or 0 if it did not.

example:

```
/* Formatted output.                                    */

prints('hello'); printtab(); prints('world');
printnl();
prints('goodbye'); printtab(); prints('world');
```

getd, geti, gets

nature: input/output functions

prototypes:

```
#include <hcifunc.h>

int geti();
double getd();
symbol gets();
```

description: These functions make a keyboard input of a `double, integer` and `symbol` variable respectively.
example:

```
/* Input of an integer variable.                        */

i = geti();
```

4.2 diskfunc

The diskfunc library provides the user with a set of simple disk I/O functions. This library is currently not available on every architecture (see A.4 for details).

The functions in the diskfunc library are accessible only if the file `diskfunc.h` is included in the source code, and if the code is compiled with the `-ldiskfunc` option.

fopen, fclose

nature: input/output functions

prototypes:

```
#include <diskfunc.h>

symbol fopen(symbol,symbol);
int fclose(symbol);
```

description: These functions open and close a file on the disk. The `fopen` function takes two symbols as argument: the name of the file to be open, and the access mode. This mode can be: `r` (read-only), `w` (overwrite), `a` (append). Its result is a symbol, called *file symbol* , that can be further used to access the newly open file (for example in an fprint function), or `NIL` if the file could not be opened.

Caution: a file symbol must never be communicated from one cell to another, neither through a channel, nor through a cell parameter. All I/O operations concerning a given file must be executed in a single cell.

The `fclose` function takes as its argument a file symbol, and closes this file. The result is 1 if the file could be closed, or 0 if not.

fprintb, fprintd, fprinti, fprints, fprintq

nature: input/output functions

prototypes:

```
#include <diskfunc.h>

int fprintb(symbol,boolean);
int fprinti(symbol,int);
int fprintd(symbol,double);
int fprints(symbol,symbol);
int fprintsq(symbol,symbol);
```

description: These functions write their second argument in the file pointed out by their first argument (a file symbol). The result is 1 if the output succeeded, or 0 if it did not.

- `fprintb` write a boolean.
- `fprintd` write a floating point number (double).
- `fprinti` write an integer.
- `fprints` write a symbol.
- `fprintsq` write a symbol between quotes ('').

examples:

```
/* Declare a file symbol fs.                         */

symbol fs;

/* Open the 'results' file for output.               */

fs = fopen('results','w');

/* Write a symbol and an integer.                    */

if (fs != NIL) {
  fprints(fs,'hello');
  fprinti(fs,i);
}

/* Close the 'results' file.                         */

fclose(fs);
```

fprinttab, fprintnl

nature: input/output functions

prototypes:

```
#include <hcifunc.h>

int printtab(symbol);
int printnl(symbol);
```

description: `fprinttab` writes a tabulation character in the file pointed out by its only argument (a file symbol); `fprintnl` writes a new-line character. The result is 1 if the output succeeded, or 0 if it did not.

example:

```
/* Declare a file symbol fs.                              */

symbol fs;

/* Open the 'results' file for output.                    */

fs = fopen('results','w');

/* Formatted output.                                      */

fprints(fs,'hello'); fprinttab(fs);
fprints(fs,'world');  fprintnl(fs);
fprints(fs,'goodbye'); fprinttab(fs);
fprints(fs,'world');

/* Close the 'results' file.                              */

fclose(fs);
```

fgetd, fgeti, fgets

nature: input/output functions

prototypes:

```
#include <diskfunc.h>

int fgeti(symbol);
double fgetd(symbol);
symbol fgets(symbol);
```

description: These functions take a file symbol as argument, and read respectively a `double, integer, symbol` variable in the corresponding file.
example:

```
/* Input of an integer variable.                    */

i = fgeti(fs);
```

4.3 hcicell

The hcicell library defines a human-computer interface cell type. This predefined `hcicell` cell type is in charge of realizing keyboard and screen I/O. Any cell can create an `hcicell` cell and send to it some objects of type integer, floating point, boolean or symbol to print, or get from it some objects of type integer, floating point or symbol read on the keyboard. This is done through a set of arrays of channels, one input and one output channel array for each type, plus a command channel indicating to the `hcicell` cell what is the action it has to do, an option channel indicating the sequence of I/O operations to be done, and a result channel indicating if the operations succeeded.

The `hcicell` predefined cells are accessible only if the file `hcicell.h` is included in the source code, and the code is compiled with the `-lhcicell` option.

The `hcicell` cell prototype given in `hcicell.h` is the following:

```
#include <hcicell.h>

typecell hcicell() {
  in int command;
  in symbol option;
  in boolean bool_in[8];
  in int int_in[8];
  in double double_in[8];
  in symbol symbol_in[8];

  out int res;
  out boolean bool_out[8];
  out int int_out[8];
  out double double_out[8];
  out symbol symbol_out[8];
}
```

An `hcicell` cell is like any other cell. In order to be used, it has to be created by a `create` request, and its channels have to be connected. An `hcicell` can print or read up to eight items in one cycle. The action to be performed by the `hcicell` cell must be sent on the `command` channel. The possible commands are the following:

PRINTI Print an integer.

PRINTB Print a boolean.

PRINTD Print a double.

PRINTS Print a symbol.

PRINTSQ Print a symbol surrounded by quotes.

PRINTTAB Print a tabulation character.

PRINTNL Print a new-line character.

GETI Get an integer.

GETD Get a double.

GETS Get a symbol.

ACTIONS Used to perform several I/O in one cycle (see below).

These commands are compatible with the `diskcell` commands, and can be sent to both kinds of cells (`diskcell` and `hcicell`).

The `hcicell` can be used in two modes: single item printing or reading, and multiple items printing or reading in one step. In the first case, only the first index `[0]` of each array of input and output channels is used. In the last case, several indexes of each array can be used, and read and write operations can be mixed.

To print a single integer, it is necessary to connect at least the `int_in[0]` channel and the `command` channel, and to send the integer on the `int_in[0]` channel and the command `PRINTI` on the `command` channel. Due to the communication mechanism, the printing will be done at the next cycle.

To get a single symbol, it is necessary to connect at least the `symbol_out[0]` channel and the `command` channel, and to send the command `GETS` on the `command` channel. The result of the input will be available in the `symbol_out[0]` channel. Due to the communication mechanism, the keyboard input will be done at the next cycle, and the result will be available at the cycle following that.

To print or get several items in one cycle, it is necessary to use the `ACTIONS` command. When `ACTIONS` is sent on the `command` channel, the `options` channel must be fed with a symbol (1 to 8 characters long) which is a series of letters using the following:

I Print an integer.

B Print a boolean.

D Print a double.

S Print a symbol.

Q Print a symbol surrounded by quotes.

T Print a tabulation character.

N Print a new-line character.

i Get an integer.

d Get a double.

s Get a symbol.

Thus, if `ACTIONS` is sent on the `command` channel, and the symbol `'SSIBiis'` is sent on the `options` channel, the `hcicell` cell will print two symbols (`SS`, print an integer (`I`), print a boolean (`B`), get two integers (`ii`), get a symbol (`s`), *provided that* the corresponding input channels `symbol_in[0]`, `symbol_in[1]`, `integer_in[0]`, `boolean_in[0]` have been fed with the values to print. Two cycles later the results of the two inputs in the channels `int_out[0]`, `int_out[1]` and `symbol_out[0]` will be available.

These I/O operations are often done using subrules (operator `-->`). It is necessary to reset the `command` channel to `NOP` after any operation, otherwise the operation would be repeated forever at each cycle, since the `command` channel would still contain the last command of the user.

The example below shows the necessary declarations and connections to the `hcicell` cell and a simple printing of an integer. Next two methods usable for a more complex sequence of combined input/outputs are shown.

examples:

```
/* Definition of a main cell using a hcicell.         */

typecell main() {

/* Necessary channels to connect to the hcicell cell. */

out int command;
out symbol options;

/* Channels supposed to be necessary in our example.  */

in  int int_in;
out int int_out;
out symbol symbol_out[2];

immat hcicell HCI;
int calc;
```

```
/* Creation of a hcicell cell.

initially ==> {
  HCI = create hcicell();

  connect command      of HCI  to command       of self;
  connect option       of HCI  to options       of self;

  connect int_in       of self to int_out[0]    of HCI;
  connect int_in[0]    of HCI  to int_out       of self;
  connect symbol_in[0] of HCI  to symbol_out[0] of self;
  connect symbol_in[1] of HCI  to symbol_out[1] of self;

  command = NOP;
}


/* Printing of a single integer.                         */

... ==> {command = PRINTI;
         int_out = 12;}
    --> command = NOP;


/* Input and usage of a single integer.                  */

... ==> {command = PRINTS;
         symbol_out[0] = 'n = ?';}
    --> command = GETI;
    --> command = PRINTNL;
    --> {command = PRINTS;
         symbol_out[0] = 'Thanks';
         calc = int_in*2;}
    --> command = NOP;


/* Same operation using ACTIONS.                         */

... ==> {command = ACTIONS;
         options = 'SiNS';
         symbol_out[0] = 'n = ?';
         symbol_out[1] = 'Thanks';}
    --> command = NOP;
    --> calc = int_in*2;
```

It is usually necessary to set the `command` channel to `NOP` after any operation because this command persists in the channel. However, this persistence can be useful in the case of a pipe-lined I/O. The example below fills an array of integers from the keyboard.

```
/* Pipe-lined input.                                        */

... ==> {command = GETI;
         index = 0;
         continue = TRUE;}

continue ==> {
   tab[index] = int_in;
   if (index < MAX)
     index += 1;
   else {
     command = NOP;
     continue = FALSE;}
}
```

The output channel `res` of the `hcicell` cell is used to know the number of I/O operations of the command which succeeded. In case of simple input or output operations, the value of `res` is 0 if the operation did not succeed, and 1 otherwise. In case a of combined I/O operation (`ACTIONS`), the I/O sequence stops at the first unsuccessful operation.

4.4 diskcell

The diskcell library defines a disk I/O interface cell type. This predefined `diskcell` cell type is in charge of realizing disk I/O. Any cell can create a `diskcell` cell and send to it some objects of type integer, floating point, boolean or symbol to write in a file, or get from it some objects of type integer, floating point or symbol read from a file. This is done through a set of arrays of channels, one input and one output array channel for each type, plus a command channel indicating to the `diskcell` cell what is the action it has to do, an option channel indicating the sequence of I/O operations to be done, and a result channel indicating if the operations went well.

The `diskcell` predefined cells are accessible only if the file `diskcell.h` is included in the source code, and the code is compiled with the `-ldiskcell` option.

The `diskcell` cell prototype given in `diskcell.h` is the following:

```
#include <diskcell.h>

typecell diskcell() {
  in int command;
  in symbol option;
  in boolean bool_in[8];
  in int int_in[8];
  in double double_in[8];
  in symbol symbol_in[8];

  out int res;
  out boolean bool_out[8];
  out int int_out[8];
  out double double_out[8];
  out symbol symbol_out[8];
}
```

A `diskcell` cell is like any other cell. In order to be used, it has to be created by a `create` request, and its channels have to be connected. A `diskcell` cell can write or read up to eight items in one cycle. Two files can be opened simultaneously by a `diskcell` cell: one read-open file, and one write-open file. Each writing command will be directed in the write-open file; each reading command will be directed in the read-open file. The action to be performed by the `diskcell` cell must be sent on the `command` channel. The possible commands are the following:

FREAD_OPEN Open a file for reading. The name of the file must be placed in the `symbol_in[0]` channel (see below).

FWRITE_OPEN Open a file for writing. The name of the file must be placed in the `symbol_in[0]` channel (see below).

FREAD_CLOSE Close the read-open file.

FWRITE_CLOSE Close the write-open file.

FCLOSE_ALL Close all the files.

FEOF Test the end of the read-open file. The boolean result is sent back in the `bool_out[0]` channel.

PRINTI Write an integer in the write-open file.

PRINTB Write a boolean in the write-open file.

PRINTD Write a double in the write-open file.

PRINTS Write a symbol in the write-open file.

PRINTSQ Write a symbol surrounded by quotes in the write-open file.

PRINTTAB Write a tabulation character in the write-open file.

PRINTNL Write a new-line character in the write-open file.

GETI Read an integer from the read-open file.

GETD Read a double from the read-open file.

GETS Read a symbol from the read-open file.

ACTIONS Used to perform several I/O in one cycle (see below).

The commands not beginning with `F` are compatible with the `hcicell` commands, and can be sent to both kinds of cells (`hcicell` or `diskcell`).

The `diskcell` can be used in two modes: single item writing or reading, and multiple items writing or reading in one step. In the first case, only the first index `[0]` of each array of input and output channels is used. In the last case, several indexes of each array can be used, and read and write operations can be mixed.

Before any read operation, a file must be read-open, by issuing a command `FREAD_OPEN` command on the `command` channel, and the file name in the channel `symbol_in[0]`. If a file is read-open and another is already read-open, the first is automatically closed.

Before any write operation a file must be write-open, by issuing a command `FWRITE_OPEN` command on the `command` channel, the file name in the channel `symbol_in[0]`, and a symbol 'w' or 'a' indicating if the file must be rewritten or appened in the `option` channel. If a file is write-open and another one is already write-open, the first one is automatically closed first.

After I/O operations, the open files should be closed, by issuing a `FREAD_CLOSE`, `FWRITE_CLOSE` or `FCLOSE_ALL` command on the `command` channel. If the I/O cell dies while some files are still open, they will be properly closed.

To print a single integer, it is necessary to connect at least the `int_in[0]` channel and the `command` channel, and to send the integer on the `int_in[0]` channel and the command `PRINTI` on the `command` channel. Due to the communication mechanism, the writing will be done at the next cycle.

To get a single symbol, it is necessary to connect at least the `symbol_out[0]` channel and the `command` channel, and to send the command `GETS` on the `command` channel. The result of the input will be available in the `symbol_out[0]` channel. Due to the communication mechanism, the file input will be done at the next cycle, and the result will be available at the cycle following the next one.

To write or get several items in one cycle, it is necessary to use the `ACTIONS` facility. When `ACTIONS` is sent on the `command` channel, the `options` channel must be fed with a symbol (1 to 8 characters long) which is a series of letters using the following:

I Write an integer.

B Write a boolean.

D Write a double.

S Write a symbol.

Q Write a symbol surrounded by quotes.

T Write a tabulation character.

N Write a new-line character.

i Read an integer.

d Read a double.

s Read a symbol.

Thus, if `ACTIONS` is sent on the `command` channel, and the symbol `'SSIBiis'` is sent on the `options` channel, the `diskcell` cell will write two symbols (`SS`) , write an integer (`I`) , write a boolean (`B`) in the write-open file, and read two integers (`ii`), read a symbol (`s`) from the read-open file, *provided that* the corresponding input channels `symbol_in[0]`, `symbol_in[1]`, `integer_in[0]`, `boolean_in[0]` have been fed with the values to write, and the two files are open. Two cycles later the results of the two inputs in the channels int_out[0], int_out[1] and symbol_out[0] will be available.

When reading a file, it is possible to know if the end of the file has been reached by sending FEOF on the command channel. At the next cycle, the channel bool_out[0] will contain TRUE if the end of file has been reached, and FALSE if not.

These I/O operations are often done using subrules (operator -->). It is necessary to reset the command channel to NOP after any operation, otherwise the operation will be repeated forever at each cycle, since the command channel will still contain the last command of the user.

The example below shows the necessary declarations and connections to the diskcell cell and a simple writing of an integer. Two methods usable for a more complex sequence of combined input/outputs follow.

examples:

```
/* Definition of a main cell using a diskcell cell.     */

typecell main() {

/* Necessary channels to connect to the diskcell cell.  */

out int command;
out symbol options;

/* Channels supposed to be necessary in our example.    */

in  int int_in;
out int int_out;
out symbol symbol_out[2];

immat diskcell DISK;
int calc;

/* Creation of a diskcell cell.                         */

initially ==> {
  DISK = create diskcell();

  connect command      of DISK  to command       of self;
  connect option       of DISK  to options       of self;

  connect int_in       of self to int_out[0]     of DISK;
  connect int_in[0]    of DISK  to int_out       of self;
```

```
  connect symbol_in[0] of DISK  to symbol_out[0] of self;
  connect symbol_in[1] of DISK  to symbol_out[1] of self;

  command = NOP;
}


/* Writing of a single integer in the file out_file.   */

... ==> {symbol_out[0] = 'out_file';
         command = FWRITE_OPEN;
         options = 'w';}
    --> {command = PRINTI;
         int_out = 12;}
    --> command = FWRITE_CLOSE;
    --> command = NOP;
```

It is usually necessary to set the `command` channel to `NOP` after any operation because this command remain in the channel. However, this fact can be useful in the case of a pipe-lined I/O. The example below fills an array of integers from the the `in_file` file, supposed large enough.

```
/* Pipe-lined input.                                   */

... ==> {symbol_out[0] = 'in_file';
         command = FREAD_OPEN;}

    --> {command = GETI;
         index = 0;
         continue = TRUE;}

continue ==> {
   tab[index] = int_in;
   if (index < MAX)
     index += 1;
   else {
     command = NOP;
     continue = FALSE;}
}
```

The output channel `res` of the `diskcell` cell is used to know the number of I/O operations of the command which have succeeded. In case of simple input or output operations, the value of `res` is thus 0 if the operation did not succeed, and 1 otherwise. In case of a combined I/O operation (`ACTIONS`), the I/O sequence stops at the first unsuccessful operation.

4.5 math

The math library provides the user with a variety of mathematical functions. These functions are accessible only if the file `math.h` is included in the source code, and if the code is compiled with the `-lmath` option.

Most of these functions are translated in calls of the corresponding functions included in the target C compiler architecture: refer to the corresponding Unix manual page. These functions cannot be redefined inside a ParCeL-1 program, even if the program does not use the math library.

The parameter ranges of these functions are those of the corresponding target C functions. If the range is not respected, the results of these functions are implementation-dependant.

Exponential and logarithmic functions

```
double exp(double);
double exp2(double);
double exp10(double);
double log(double);
double log2(double);
double log10(double);
```

Trigonometric functions

```
double sin(double);
double cos(double);
double tan(double);
double asin(double);
double acos(double);
double atan(double);
double atan2(double,double);
double sinh(double);
double cosh(double);
double tanh(double);
```

Rounding functions

```
double ceil(double);
double floor(double);
double rint(double);
int iceil(double);
int ifloor(double);
int irint(double);
```

These three last functions (`iceil`, `ifloor`, `irint`) return the same value as `ceil`, `floor` and `rint`, but in an integer format. If the result is too large to fit in the integer format, the result will be implementation-dependant.

Miscellaneous functions

```
double fabs(double);
double pow(double,double);
double sqrt(double);
double fmod(double,double);
```

4.6 std

The std library provides the user with some useful common functions to access the command line parameters, to cast types, and to generate random numbers. These functions are accessible only if the file `std.h` is included in the source code, and the code is compiled with the `-lstd` option.

argc, argv

nature: command line parameters functions

prototypes:

```
#include<std.h>

int argc();
symbol argv(int);
```

description: `argc` returns the number of items on the command line, that is, the number of parameters plus one (the program name). `argv` takes an integer n as parameter and returns the value of the n^{th} command line parameter as a symbol. `argv(0)` returns the program name. These two functions are currently not available on `icc_nt` target architecture (see A.4 for details).

clock

nature: report time used

prototypes:

```
#include<std.h>

double clock();
```

description: `clock` returns a time in seconds. The difference between the results of two calls to `clock()` is the ellapsed time between the two calls. On certain architectures, this function may return imprecise results. In any case, too small durations shall not be measured.

rand, srand

nature: random number generation functions

prototypes:

```
#include<std.h>

int rand();
void srand(int);
```

description: `rand` returns an integer comprised between 0 and `RAND_MAX`. `RAND_MAX` is defined in `<std.h>`. `srand` is used to initialize the random number generator with any integer. `srand` can be used to make sure the same sequence of random numbers will be produced at each running of the program.

sytoi, sytod

nature: type cast functions

prototypes:

```
#include<std.h>

int sytoi(symbol);
double sytod(symbol);
```

description: `sytoi` converts an `integer` into a `symbol`. If the symbol cannot be converted, `sytoi` returns 0. `sytod` converts a `double` into a `symbol`. If the symbol cannot be converted, `sytod` returns 0.

examples:

```
...
int i;
double d;
symbol s;

/* Cast a symbol to an integer.                    */

i = sytoi('12');

/* Cast a symbol to a double.                      */

d = sytod(s);
```

Appendix A

Compiler installation

A.1 Compilation process

The compilation of a ParCeL-1 program is done in two main steps: first, the ParCeL-1 compiler compiles ParCeL-1 source code into C source code. Then a C compiler generates the executable code for the target architecture.

The ParCeL-1 compiler can be installed on a variety of architectures, all running under the Unix operating system: Sun 4 (SparcStation), Linux PC, DEC Alpha and HP 700 S 9000 workstations, Cray Y-MP. It is necessary to generate a ParCeL-1 compiler for each architecture on which ParCeL-1 will be used *(compilation architecture)*. Several different compilers may coexist for different compilation architectures (e.g., one for Sun 4 and one for Linux), and the installation process will insure that the correct compiler will be selected for any of the compilation architectures.

A.2 Cross compilation

A given compilation architecture can produce executable code for several different *target architectures* (cross-compilation), provided that a C compiler producing code for the target architecture is available on the compilation architecture. For example, a Sun4 compilation architecture can produce code for Sun, Parsytec mtmsun board and T-node target architectures. On the contrary, an Alpha compilation architecture can produce code only for Alpha target architecture. Thus, given a compilation architecture, it is also necessary to compile one set of ParCeL-1 libraries for each target architecture.

The ParCeL-1 compiler has currently been proven to be successful in the following cross-compilation combinations:

Compilation architecture	*Target architecture*
Sun4	Sun4, Parsytec mtmsun boards, T-Node
Linux PC	Linux PC
DEC Alpha	DEC Alpha
HP 9000 S 700	HP 9000 S 700
Cray Y-MP	Cray T3D

A.3 Supported architectures

ParCeL-1 has been tested on the following compilation and target architectures:

Hardware	*Operating system*	*cc version*	*gcc version*
Sun 4	SunOS 4.1.3		2.5.7
Linux PC	Linux 1.1.54		2.5.8
DEC Alpha	OSF/1 V2.0		2.5.4
HP 9000 S 700	HP-UX 9.0		2.5.8

ParCeL-1 has been tested on the following target architectures:

Hardware	*C compiler*
T-node	Inmos C 2.01.10
Parsytec mtmsun board	Par.C 1.32
Cray T3D	C89

A.4 Supported functionalities by architecture

Some of the features of ParCeL-1 are available only on some targets, listed in the table below. Any feature not in the table is available on every target.

target	`-g`	`-prof`	`-timing`	hcifunc diskfunc	`argc()` `argv()`
Workstation	x	x	x	x	x
`parc_1t`	x		x	x	x
`parc_nt`	x		x	x	x
`icc_1t`	x		x	x	
`icc_nt`					
`c89_t3d_np`			x	x	x

A.5 Compiler installation

To install the ParCeL-1 compiler, observe the following procedure:

1. Uncompress and untar the `parcel.tar.Z` file. This will create a `parcel` directory under the current directory.

2. Define an environment variable `PARCEL` giving the complete path to the new `parcel` directory by typing `setenv PARCEL /...../parcel`.

3. Complete the `CONFIG.H` file of the `parcel` directory with your preferred options (the options are documented in the file).

4. Execute the target `preinstall` of the makefile, by typing `make preinstall`.

5. Install the VCM-1 libraries for each target architecture for which you want to generate executable code. The available makefile targets are: vcm_lib_cc_sun4, vcm_lib_gcc_sun4, vcm_lib_cc_alpha, vcm_lib_gcc_alpha, vcm_lib_cc_i386-linux, vcm_lib_gcc_i386-linux, vcm_lib_cc_hp9000s700, vcm_lib_gcc_hp9000s700, vcm_lib_parc_1t, vcm_lib_parc_nt, vcm_lib_icc_1t, vcm_lib_icc_nt, vcm_c89_lib_t3d_1p, vcm_lib_c89_t3d_np.

6. Install the ParCeL-1 libraries for each target architecture for which you want to generate executable code. The available makefile targets are: parcel_lib_cc_sun4, parcel_lib_gcc_sun4, parcel_lib_cc_alpha, parcel_lib_gcc_alpha, parcel_lib_cc_i386-linux, parcel_lib_gcc_i386-linux, parcel_lib_cc_hp9000s700, parcel_lib_gcc_hp9000s700, parcel_lib_parc_1t, parcel_lib_parc_nt, parcel_lib_icc_1t, parcel_lib_icc_nt, parcel_lib_c89_t3d_1p, parcel_lib_c89_t3d_np.

7. Install one ParCeL-1 compiler for each compilation architecture. Available makefile targets are: parcel_cc_sun4, parcel_gcc_sun4, parcel_cc_alpha, parcel_gcc_alpha, parcel_cc_i386-linux, parcel_gcc_i386-linux, parcel_cc_hp9000s700, parcel_gcc_hp9000s700, parcel_c89_cray-unicos.

8. In your `.cshrc` file, add the line: `source /...../parcel/shell/parcel.env`.

9. Source your new `.cshrc` file.

To complete ParCeL-1 installation for new target or compilation architectures, it is necessary to execute the corresponding make target: `vcm...` for the virtual machine (target architectures) `lib....` for the libraries (target architectures) and `parcel....` for the compiler (compilation architectures).

If the compiler is already installed and a user wants to run it, the user has to add the line `source /...../parcel/shell/parcel.env` in his `.cshrc` file, and then source it.

Appendix B

Error messages

B.1 Error referencing

This chapter gives the list of all the ParCeL-1 possible errors. There are two main types or errors: compilation errors and run-time errors. There are four types of compilation errors (labelled C, SC, PC and IC):

- C errors: program-concerned errors, especially syntax errors and semantic errors. The source code must be modified to correct these errors.
- SC errors: system-concerned errors. These errors are concerned with the underlying operating system (for example, problems with read or write permissions). It is not necessary to modify the the source code to correct these errors.
- PC errors: pre-processor errors. These errors are due to bad preprocessing directives in the source code.
- IC errors: internal errors. These errors are not supposed to occur, and may be due to a bug in the ParCeL-1 compiler. Please report to the authors (electronic mail: *vialle@ese-metz.fr*).

There are three types of run-time errors (labelled RR, RC and RF):

- RR errors: the errors concerning the request coherence.
- RC errors: the errors concerning the checkings done when the `-g` option is used during compilation.
- RF errors: the fatal errors leading to the stop of the program.

During compilation, each error displayed will indicate the source file and the line where the error appeared. The compiler will indicate all the errors it can find in only one compilation.

During run-time, RR errors will only appear on sequential architectures, and RC errors will only appear if the running program has been compiled with the `-g` flag. RF error will appear in any case.

B.2 Error levels

Each request run-time error (referenced RR) is attributed an *error level*. This level is an integer between 1 and 5; the higher the level, the more serious the error. The `-w` and `-h` options allow the user to set the level of errors about which he wants to be warned or to halt the program (see 2.5.3).

The meaning of the error levels are the following:

level 1 These errors are inoffensive. When they appear, the running of the program is not affected. Awakening an already awake cell will produce a level 1 error.

level 2 These errors are also inoffensive, but regard dead cells, thus can be more serious, since a dead cell should not be addressed. Sleeping a dead cell will produce a level 2 error.

level 3 These errors mean that the requested action could not be executed, and thus may hamper the future running of the program. Disconnecting channels not connected to each other will produce a level 3 error.

level 4 Unused.

level 5 These errors mean that the requested action could not be executed, and will certainly trouble the future running of the program. Connecting a channel to a dead cell will produce a level 5 error.

Any RC or RF error will stop the program. A RR error will stop the program only if its level is greater then the `-h` level defined at the compilation.

B.3 Compilation error messages

The compilation error messages are most often self-descriptive. In this section we only describe in detail the complex messages.

C01 `<private variable> only accessible in cell environment.`
A private variable such as `self` is used outside a cell, and thus has no meaning.

C02 `Error on <array access> on array <array name>, index should be integer.`
A non-integer index has been used to access an array.

C03 `Argument <identifier> has illegal type for <function|cell>.`
The type of the argument does not match the declared type for this argument in this function.

C04 `Operator <operator> has incompatible type arguments.`
A given operator can operate only on certain types of arguments. See reference manual for the given operator.

C05 `Channels <identifier> and <identifier> have incompatible types.`
Two channels can be connected with each other only if they are of the same type.

C06 `<identifier> forbidden in array declaration.`
An array declaration can contain only integer constants or simple arithmetic operators.

C07 `Dimension <number> of array <identifier> is not strictly positive.`
A negative index has been used to access an array.

C08 `Argument <identifier> of operator <operator> should be writable.`
Attempt to modify a read-only variable such as `self`.

C09 `<identifier> should be a variable of type immat.`

C10 `Argument type of 'return' incompatible with function type.`
The type of the result returned by the function does not match the type of the function.

C11 `Argument in return only allowed in non-void functions.`
The only possible return statement in a `void` function is `return()`.

C12 `'void' is not a legal type for a parameter.`
Only a function can be of type `void`.

C13 `'void' is not a legal type for a variable.`
Only a function can be of type `void`.

C14 `Division by zero.`

C15 `Variable <identifier> of <cell|function> <identifier> should be a channel.`
An instruction or a request (such as `connect`) needs a channel and is given something else.

C16 `Variable <identifier> of <cell|function> <identifier> should be an output channel.`
An instruction or a request (such as `connect`) needs an output channel and is given something else.

C17 `Missing 'main' cell type.`
It is necessary to declare a `main` cell type in any program.

C18 `<identifier> used but not defined.`
An undeclared function or cell has been used; only its prototype is known. Make sure the good library has been linked to (with the `-l` compiler option).

C19 `<identifier> undeclared.`
An undeclared variable has been used.

C20 `Input channel <identifier> undeclared in cell <identifier>.`
An undeclared input channel has been used in the given cell.

C21 `Output channel <identifier> undeclared in cell <identifier>.`
An undeclared output channel has been used in the given cell.

C22 `<cell|function> already defined in <location>.`
Multiple declarations of the same cell or function.

C23 `Cell <identifier> declared with a different number of input channels in <location>.`
A cell has already been declared elsewhere with a different number of input channels.

C24 `Cell <identifier> declared with a different number of output channels in <location>.`
A cell has already been declared elsewhere with a different number of output channels.

C25 `Initially rule already declared.`
A cell can contain only one `initially` rule.

C26 `Finally rule already declared.`
A cell can contain only one `finally` rule.

C27 `<cell|function> <identifier> already declared in <location>.`
Multiple declarations of a function or a cell.

C28 `<cell|function> <identifier> declared with another arity in <location>.`
Incompatible declarations of a cell or a function.

C29 `<variable|function> declared with another type in <location>.`
Incompatible declarations of a variable or a function.

C30 `Array <identifier> declared with another dimension in <location>.`
Incompatible declarations of two arrays.

C31 `Parameter <number> of <cell|function> declared with another type in <location>.`
Type mismatch in several declarations of the same cell or function.

C32 `Dimension <number> of array <identifier> declared with a different value in <location>.`
Incompatible declarations of an array.

C33 `Too many errors, compilation aborted.`
The compiler found too many errors and aborted the compilation. Correct the errors that have already appeared.

C34 `Unexpected end of file in a symbol.`
The source code does not properly end.

C35 `Unexpected end of line in a symbol.`
The source code does not properly end.

C36 `Unexpected end of file in a comment.`
The source code does not properly end.

C37 `Syntax error on or before <item>.`
A syntax error, such has a missing semicolon, appeared in the given file on the given line. Check the source code.

C38 `Inline library function <function> has another prototype.`
The prototype of an inline function does not match its definition.

C39 `Inline library function <function> cannot be redefined.` An inline function cannot be redefined.

SC01 `Shell execution impossible.`
ParCeL-1 cannot execute a Unix shell needed for the compilation process.

SC02 `Compilation aborted: system error or target compilation uncompatible with this installation.`
A Unix system error has occured, or ParCeL-1 cannot compile for the requested target, possibly because the corresponding ParCeL-1 libraries have not been installed.

SC03 `Not enough memory.`
ParCeL-1 cannot compile on this machine because it lacks memory.

SC04 `Unable to create file: <file>.`
ParCeL-1 needs to create a private file, but cannot, possibly because of wrong access permissions.

SC05 `Unable to read file: <file>.`
ParCeL-1 needs to read a file, but cannot, possibly because of wrong access permissions.

SC06 `Environment variable PARCEL undefined.`
The path to ParCeL-1 root directory should be contained in the `PARCEL` environment variable (see the compiler installation section).

SC07 `Unable to access tmp directory: <directory name>.`
The temporary directory defined in the `CONFIG.H` file cannot be accessed (see the compiler installation section).

SC08 `Default target <target name> in CONFIG.H unknown.`
The default target defined in the `CONFIG.H` file does not exist (see the compiler installation section).

SC09 `Option -g not supported by target <target name>.`
The `-g` option is available only for sequential architectures, see A.4 for details.

SC10 `Option -prof not supported by target <target name>.`
Profilers are not available on every architecture: see A.4 for details.

SC11 `Option -timing not supported by target <target name>.`
Timers are not available on every architecture: see appendix A.4 for details.

SC12 `Error in command line on <item>.`
There is a syntax error in the ParCeL-1 command line, possibly a non existing option, a bad or omitted value for an option, or bad file specification.

SC13 `Wrong command line, enter parcel -help for usage.`
The command line is not well formed.

PC01 `Compilation aborted: errors detected by preprocessor.`
Check the preprocessor directives in your source code.

IC `Internal error <number>.`
An unexpected error has occured. Please report to the authors (electronic mail: *vialle@ese-metz.fr*).

B.4 Compilation warning

W01 `Warning: truncated symbol: <symbol>.`
This is not an actual error, but only a warning. A symbol can have 1 to 8 characters long, and longer symbols are truncated during compilation. This warning can be avoided by compiling with the `-s` flag.

B.5 Run-time error messages

RC01 `Division by zero.`

RC02 `Index <index name> of array <array name> out of range: <index value> (max <index maximum value>).`
An index has appeared in a given array which does not match the definition of this area (too low or too high).

RC03 `Wrong cell immatriculation in a cell of type <cell name>.`
An immatriculation referring to a non-existing cell has been used.

RF01 `Illegal cell immatriculation used.`
A cell immatriculation corresponding to a non-existing cell has been used.

RF02 `Impossible request.`
An impossible request has been attempted, such as awakening an already awake cell. This fatal error is produced if a RR error has been produced whose level is higher than the `-h` level defined during compilation.

RF03 `One request table is full.`
ParCeL-1 has no more room for more requests. Recompile the program with an increased `-request` value (see section 2.5.3).

RF04 `Tables of channels full.`
ParCeL-1 has no more room for any new channel connection. Recompile the program with an increased `-channel` value (see section 2.5.3).

RF05 `Table of cell immatriculations full.`
ParCeL-1 has no more room for any new cell creation. Recompile the program with an increased `-cell` value (see section 2.5.3).

RF06 `Full stack.`
ParCeL-1 has no more room in its internal stack. Recompile the program with an increased `-stack` value (see section 2.5.3).

RF07 `Insufficient memory.`
ParCeL-1 is out of memory. The program cannot be run on this machine.

RR01 (level 1) `Awake attempted on an already awake cell.`

RR02 (level 5) `Awake attempted on a dead cell.`

RR03 (level 5) `Connect attempted with a dead reader cell.`

RR04 (level 5) `Connect attempted with a dead writer cell.`

RR05 (level 1) `Connectiffree attempted on a connected input channel.` The `connectiffree` request operates only on unconnected channels. If the target channel is already connected, this warning is issued and the program execution goes on.

RR06 (level 1) `Disconnect attempted with an already disconnected input channel.`

RR07 (level 1) `Disconnect attempted with an already disconnected output channel.`

RR08 (level 2) `Disconnect attempted with an output channel of a dead cell.`

RR09 (level 2) `Disconnect attempted with an input channel of a dead cell.`

RR10 (level 3)`Disconnect attempted with channels not connected to each other.`

RR11 (level 1) `Kill attempted on an already dead cell.`

RR12 (level 1) `Sleep attempted on a sleeping cell.`

RR13 (level 2) `Sleep attempted on a dead cell.`

Glossary

Cell. Virtual processor, executing its program and communicating with some other cells.

Channel. Variable shared by several cells. See *output channel* and *input channel*.

Connection. Act of linking an input channel to an output channel. Once connected, the input channel will have the same value as the output channel.

Compilation architecture. The architecture on which the ParCeL-1 compiler runs.

Cycle. Update of all the cells. An update can consist of executing the *action* part of a selectable rule, or doing nothing, if the cell is asleep or if no rule is selectable.

Immatriculation. Reference to a given cell in order to address this cell, for example, to connect it.

Input channel. When connected, receives a value from an output channel.

Main. Type of the first created cell when a ParCeL-1 program is run. A `main` cell must be present in every program.

Output channel. When connected, sends a value to an input channel.

Request. Command sent to the VCM-1 which concerns the topology of the network of cells. A request needs one cycle to be executed, and thus the result is effective only at the next cycle.

Rule. `<condition> ==> <action>`. Determines the behavior of the cell.

Subrule. `<condition> -(i)-> <action>`. After a given rule, indicates how many cycles have to be run before executing *action*.

Sequence. Set of subrules of the same rule determining an unbreakable sequence of actions to execute.

Target architecture. The architecture on which the user's program compiled with ParCeL-1 runs.

VCM-1. Virtual Cellular Machine. Underlying computational model of ParCeL-1, featuring cyclic execution, channel data sharing, etc.

Acknowledgements

The authors wish to thank:

Prof. Claude Lhermitte and Prof. Claude Timsit for fruitful discussions that shaped the software;
Martial Bardot, Pierre-Marie Borgeal, Sylvain Corré, Stéphane Gay, Franck Louveau, Daniel Méketyn, Francois Parmentier, Marc Scalzolaro and Frédéric Wolff for their active participation in the project;
Brandon Nelson, Bradley Orfall, Henry Schneider, Mark Walmsley and Richard Washington for their help in proofreading this document.

Our gratitude also to Patrick Mercier for his technical assistance, and to our respective laboratories and research teams for their constant support to this project.

ParCeL-1 development has been mainly supported by SUPÉLEC. Its revision and the preparation of this manual have been supported by CRIN-INRIA Lorraine, EPFL and SUPÉLEC.

Bibliography

[Bar94] M. Bardot. *Implantation et utilisation de la MCV et de ParCeL sur T-Node*. DEA thesis, Université de Nancy I and Supélec-Metz, France, 1994. In French.

[CH90] T. Cornu and J.P. Haton. An actor language for connectionnism based on cellular automata. In *International Neural Network Conference, INNC90*, Paris, July 1990.

[Cor92] T. Cornu. *Machine Cellulaire Virtuelle : Définition, Implantation et Exploitation*. PhD thesis, Université de Nancy I, CRIN INRIA Lorraine, Nancy, France, October 1992. In French.

[CV93] T. Cornu and S. 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*, Leeds, England, apr 1993. Oxford University Press. in press.

[Gay92] S. Gay. *Réalisation d'un Système de Communication Multiprocesseurs Optimisé*. DEA thesis, Université de Nancy I and Supélec Metz, France, 1992. In French.

[GV92] S. Gay and S. Vialle. Analyse des divers parallélismes du transputer. *Lettre du transputer et des calculateurs distribués*, (15):27–42, sep 1992. In French.

[Lal91] Y. Lallement. *Machine Cellulaire Virtuelle et Intelligence Artificielle*. DEA thesis, Université de Nancy I and Supélec Metz, France, 1991. In French.

[LCV93] Y. Lallement, T. Cornu, and S. Vialle. An abstract machine for implementing connectionnist and hybrid systems on multi-processor architectures. In H. Kitano, V. Kumar, and C.B. Suttner, editors, *Second International Workshop on Parallel Processing for Artificial Intelligence*, Machine Intelligence and Pattern Recognition. Elsevier Science Publishers, 1993.

[LCV95] 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*, Montreal, Canada, 1995.

[Par93] F. Parmentier. *Implantation d'un réseau de neurones sur une machine cellulaire virtuelle.* DEA thesis, Université de Nancy I and Supélec-Metz, France, 1993. In French.

[Wol94] F. Wolff. *Répartition dynamique de charge dans la machine cellulaire virtuelle.* DEA thesis, Université de Nancy I and Supélec-Metz, France, 1994. In French.

These papers are available in PostScript format through anonymous ftp on ftp.ese-metz.fr, under the directory /pub/reports/info, or through WWW: http://www.ese-metz.fr/info/project/parcel/parcel.html

Index