P^3L: The Pisa Parallel Programming Language (Ver. 1.0)

M. Danelutto°* & S. Orlando°* & S. Pelagatti°

* Pisa Science Center	° Dipartimento di Informatica
Hewlett Packard	Università di Pisa
Corso Italia 115	Corso Italia 40
56125 PISA	56125 PISA
Italy	Italy

October 7, 1991

Abstract

The syntax, semantics and pragmatics of the P^3L programming language are described here. The version of the language discussed is the P^3L version 1.0. Some features have been left out that will appear in future versions of the language. The P^3L described here uses C++ has "host language". However, many of the arguments and topics discussed in the report apply independently of the host language chosen for the implementation.

Contents

List of Figures

Chapter 1

Syntax

P^3L is made up of a sequential component plus a parallel component. The sequential component will be taken as much as possible close to a classical imperative programming language to enhance software reusability. The classical imperative programming language used as basis for this P^3L version will be called henceforth "host language".

1.1 Sequential part

The host sequential language adopted for this implementation of P^3L is a strict subset of the C++ programming language.

This host language provides syntax and semantics for the procedural description of the sequential portions of the code. The types exploited in the parallel part of P^3L can be mapped to the types of the host language.

1.1.1 P^3L data types

The P^3L data types can be subdivided into *basic* and *compound* data types. The set of basic types ($\mathcal{B}$) is defined as follows:

$$\mathcal{B} \equiv \{integer, boolean, real, character\}$$

These data types are mapped to host language data types as follows:

- integers, corresponding to C++ `int` data type;
- reals, corresponding to C++ `float` data type;
- characters, corresponding to C++ `char` data type;
- boolean, corresponding to C++ `int` data type. Booleans can have only values in the set {`TRUE`,`FALSE`}.

The set of compound data types ($\mathcal{T}$) is defined as follows:

$$\forall t \in \mathcal{B} \;\Rightarrow\; t \in \mathcal{T} \tag{1.1}$$
$$\forall t \in \mathcal{T},\; n \in \mathcal{N}^+ \;\Rightarrow\; t[n] \in \mathcal{T} \tag{1.2}$$
$$\forall t_1, \ldots, t_k \in \mathcal{T} \;\Rightarrow\; \langle t_1, \ldots, t_k \rangle \in \mathcal{T} \tag{1.3}$$

nothing else is in $\mathcal{T}$.

Pointers or class data types can be used internally to the sequential portions of code, e.g. in the sequential code of a farm worker or of a pipeline stage, where the *farm* and the *pipeline* are two important parallel constructors of P^3L language.

The compound data types built applying the rule 1.2 are called *array types*, while the compound data types defined by applying the rule 1.3 are called *structure types*.
We can refer to the dimensions of an array type and to the lengths of these dimensions. For instance, $integer[4][6]$ is an array type having two dimensions, while the lengths of its dimensions is 4 and 6, respectively.
Moreover, we can refer to the *fields* of a structure type. For instance, $\langle integer,\ character\rangle$ has two fields: the former has *integer* type, while the latter has *character* type.
These data types are mapped to host language data types as follows:

- array are mapped to C++ arrays, augmented with some new features,
- structures are mapped to C++ `struct`s.

All the data types just presented can be declared **`readonly`**. The syntax for this declaration is:

```
<readonly decl>::= readonly <host language declaration>
```

and it should be used in all those cases where it is known that the data that we are going to define are not modified by the computations.

In order to achieve portability, no assumptions on data type sizes can be made as well as any kind of implicit data type conversion, nor assumptions on the memory ordering have to be made (little/big endian).

We define now some concepts that will be useful in the following to describe P^3L syntax.
The notion of type *equality* can be defined in a formal way as follows.

Definition 1.1 *$t_i \in \mathcal{T}$ and $t_j \in \mathcal{T}$* are the same type *(and we write $t_i \asymp t_j$) iff one of the following formulas holds:*

$$t_i \in \mathcal{B} \ \wedge \ t_j \in \mathcal{B} \ \wedge \ t_i = {}^1 t_j$$

$$(\exists t_i^1, t_j^1 \in \mathcal{T},\ n_1, n_2 \in \mathcal{N}^+ \ \mid \ t_i = t_i^1[n_1] \wedge t_j = t_j^1[n_2]) \ \wedge \ t_i^1 \asymp t_j^1$$

$$(\exists t_i^1, t_j^1, \ldots, t_i^k, t_j^k \in \mathcal{T} \ \mid \ t_i = \langle t_i^1, \ldots, t_i^k\rangle \wedge t_j = \langle t_j^1, \ldots, t_j^k\rangle) \ \wedge \ (\forall s \in [1,k] \ \ t_i^s \asymp t_j^s)$$

□

The Def. 1.1 regards two arrays as equal if they have the same number of dimensions, but different lengths. The following is the notion of strong equality, which also considers the lengths of the dimensions of the arrays:

Definition 1.2 *$t_i \in \mathcal{T}$ and $t_j \in \mathcal{T}$* are identical *(and we write $t_i \equiv_\tau t_j$) iff one of the following formulas holds:*

$$t_i \in \mathcal{B} \wedge t_j \in \mathcal{B} \wedge t_i = t_j$$

$$(\exists t_i^1, t_j^1 \in \mathcal{T},\ n \in \mathcal{N}^+ \ \mid \ t_i = t_i^1[n] \wedge t_j = t_j^1[n]) \ \wedge \ t_i^1 \equiv_\tau t_j^1$$

$$(\exists t_i^1, t_j^1, \ldots, t_i^k, t_j^k \in \mathcal{T} \ \mid \ t_i = \langle t_i^1, \ldots, t_i^k\rangle \wedge t_j = \langle t_j^1, \ldots, t_j^k\rangle) \ \wedge \ (\forall s \in [1,k] \ \ t_i^s \equiv_\tau t_j^s)$$

□

Thus, two types are said to be *identical* if and only if they are the *same type* and they corresponding array dimensions have equal lengths.

Now we define the notions of *proper* subtyping.

Definition 1.3 *$t_i \in \mathcal{T}$ is a* proper subtype *of $t_j \in \mathcal{T}$ (and we write $t_i \prec_p t_j$) iff one of the following formulas holds:*

$$t_i \in \mathcal{B} \wedge t_j \in \mathcal{B} \wedge t_i = t_j$$

[1] The symbol = stands for syntactical equality

$$(\exists t_k \in \mathcal{T},\ \{n_1, \ldots, n_h\} \in \mathcal{N}^+ \quad |$$
$$(t_j = t_k[n_t]_{t=1}^{h} \quad \wedge \quad (\exists k \in [1,h] \ \ | \ \ t_i \equiv_{\mathcal{T}} t_k[n_t]_{t=1}^{k-1}[n_t]_{t=k+1}^{h})^2)$$

$$(\exists t_1, \ldots, t_n \in \mathcal{T} \quad | \quad t_j = \langle t_1, \ldots, t_n \rangle \quad \wedge \quad (\exists k \in [1,n] \mid t_i \equiv_{\mathcal{T}} t_k))$$

□

In other words, t_i is a proper subtype of t_j if

- t_j and t_i are two identical basic types,
- t_j is a *structure* and t_i is the type of one of its fields,
- t_j is an *array*, while t_i has one dimension less than t_j and the lengths of the other dimensions is unchanged. In case t_j is an array having only one dimension, t_i may be a basic type or a structure.

The relationship $\prec_p$ introduces a partial order on the set of types T. The following relationship, called *weak* subtyping, is the transitive closure of $\prec_p$:

Definition 1.4 *$t_i \in \mathcal{T}$ is a* weak subtype *of $t_j \in \mathcal{T}$ (and we write $t_i \prec_w t_j$) iff one of the following formulas holds:*

$$t_i \prec_p t_j$$
$$(\exists t_k^1, \ldots, t_k^n \in \mathcal{T} \ \ | \ \ t_i \prec_p t_k^1 \prec_p t_k^2 \prec_p \ldots \prec_p t_k^n \prec_p t_j)$$

□

Let us define some other useful concepts that will be used in the following part of this report. Definition 1.5 formalizes the concept of compound type having the same structure.

Definition 1.5 *$t_i \in \mathcal{T}$* has the same data structure of *$t_j \in \mathcal{T}$ (and we write $t_i \mathcal{H} t_j$) iff one of the following formulas hold:*

$$t_i \in \mathcal{B} \wedge t_j \in \mathcal{B}$$
$$(\exists t_i^1, t_j^1 \in \mathcal{T},\ n_1, n_2 \in \mathcal{N}^+ \ \ | \ \ t_i = t_i^1[n_1] \wedge t_j = t_j^1[n_2]) \ \wedge \ t_i^1 \mathcal{H} t_j^1$$
$$(\exists t_i^1, t_j^1, \ldots, t_k^i, t_j^k \in \mathcal{T} \mid t_i = \langle t_i^1, \ldots, t_i^k \rangle \wedge \ t_j = \langle t_j^1, \ldots, t_j^k \rangle) \ \wedge \ (\forall s \in [1,k] \ t_i^s \mathcal{H} t_j^s)$$

□

Using definition 1.5, we can define the notion of equivalence between the structures of different pairs of types as follows:

Definition 1.6 *$t_i \prec_w t_j$* weakens in the same way of *$t_i' \prec_w t_j'$ iff*

$$t_i \mathcal{H} t_i' \wedge t_j \mathcal{H} t_j'$$

and we write $(t_i \prec_w t_j) \nabla \ (t_i' \prec_w t_j')$. □

Now we introduce the concept of a *covering slice*, which is related to the ability to partition an n-dimensional array in a number of n-dimensional subarrays.

Definition 1.7 *$t_i \in \mathcal{T}$ is a* covering slice *of $t_j \in \mathcal{T}$ (and we write $t_i \lhd_{cov} t_j$) iff*

$$t_i, t_j \in \mathcal{B} \wedge t_i = t_j$$

$$(\exists t_i^1, t_j^1, \ldots, t_i^k, t_j^k \in \mathcal{T} \mid t_i = \langle t_i^1, \ldots, t_i^k \rangle \wedge t_j = \langle t_j^1, \ldots, t_j^k \rangle) \ \wedge \ t_j \equiv_{\mathcal{T}} t_i$$

[2] $[n_t]_{t=1}^{h}$ is used as shorthand for $[n_1][n_2]\ldots[n_h]$

$$(\exists t_i^1, t_j^1 \in \mathcal{T},\ n_1, n_2 \in \mathcal{N}^+ \ \mid\ t_i = t_i^1[n_1],\ t_j = t_j^1[n_2]) \wedge$$
$$\wedge\ \ (n_1 \leq n_2)\ \wedge\ (n_2 \bmod n_1 = 0)\ \ \wedge\ \ t_i^1 \lhd_{cov} t_j^1$$

□

For instance, the type $integer[2][2]$ is a covering slice of the type $integer[10][10]$.

The following definition introduces the concept of partitioning compond data types by means of data subtypes.

Definition 1.8 $t_i \in \mathcal{T}$ data partitions by subtyping $t_j \in \mathcal{T}$ *(and we write* $t_i \sharp_{typ} t_j$*) iff*

$$t_i, t_j \in \mathcal{B} \wedge t_i = t_j$$
$$(\exists t_i^1, t_j^1, \ldots, t_i^k, t_j^k \in \mathcal{T} \mid t_i = \langle t_i^1, \ldots, t_i^k \rangle \wedge t_j = \langle t_j^1, \ldots, t_j^k \rangle)\ \wedge\ t_j \equiv_\tau t_i$$
$$(\exists t_k \in \mathcal{T},\ \{n_1, \ldots, n_h\} \in \mathcal{N}^+\ \mid\ t_j = t_k[n_t]_{t=1}^h)\ \wedge\ ((t_i \equiv_\tau t_k)\ \vee\ (t_k \prec_w t_i \prec_w t_j))$$

□

At last, we can give the definition that considers the two ways to cover a given array type, i.e. by subtyping or by covering slice.

Definition 1.9 $t_i \in \mathcal{T}$ datapartitions $t_j \in \mathcal{T}$ *(and we write* $t_i \sharp t_j$*) iff*

$$t_i \sharp_{typ} t_j\ \ \vee\ \ t_i \lhd_{cov} t_j$$

□

The following definition extends the relationship of datapartitiong to the various fields of a structure.

Definition 1.10 $t_i \in \mathcal{T}$ *is a* substruct-datapartion *of* $t_j \in \mathcal{T}$ *(and we write* $t_i \sharp_S t_j$*) iff*

$$(\exists t_i^1, \ldots, t_i^k, t_j^1, \ldots, t_j^k \in \mathcal{T} \mid t_i = \langle t_i^1, \ldots, t_i^k \rangle \wedge t_j = \langle t_j^1, \ldots, t_j^k \rangle)$$
$$\wedge\ (\forall h \in [1, k]\ :\ t_i^h \sharp t_j^h)$$

□

Now we formalize the relationship of *substructure* between two structured types.

Definition 1.11 $t_i \in \mathcal{T}$ *is a* substructure *of* $t_j \in \mathcal{T}$ *(and we write* $t_i \subseteq_S t_j$*) iff*

$$(\exists t_i^1, \ldots, t_i^k, t_j^1, \ldots, t_j^n \in \mathcal{T} \mid t_i = \langle t_i^1, \ldots, t_i^k \rangle \wedge t_j = \langle t_j^1, \ldots, t_j^n \rangle)\ \ \wedge$$
$$\wedge\ \ (k \leq n)\ \ \wedge\ \ (\exists \mathcal{M} : [1, k] \rightarrow [1, n]\ (injective) \mid \forall s \in [1, k] : t_i^s \equiv_\tau t_j^{\mathcal{M}(s)})$$

□

Notice that the relationship $\subseteq_S$ cannot be expressed in terms of $\prec_w$. The latter can only relate a structured type to the types of its fields, while $\subseteq_S$ relates two different structures where one forgets some of the fields of the other.

Supposing that both $t_i, t_j \in \mathcal{T}$ are structures, and that $t_i \subseteq_S t_j$, we can formally define the *complement* of t_i in t_j.

Definition 1.12 *Let* $t_i = \langle t_i^1, \ldots, t_i^k \rangle \in \mathcal{T}$ *and* $t_j = \langle t_j^1, \ldots, t_j^n \rangle \in \mathcal{T}$ *be two structures s.t.*
$k \leq n \quad \wedge \quad t_i \subseteq_S t_j$.
Thus, there exists a function (see Def. 1.11) $\mathcal{M}_{sub} : [1,k] \rightarrow [1,n]$ *(injective) s.t.*
$\forall s \in [1,k] : t_i^s \equiv_\tau t_j^{\mathcal{M}_{sub}(s)}$.

$t_j - t_i \; = \; \langle t_{j-i}^1, \ldots, t_{j-i}^{n-k} \rangle \in \mathcal{T}$ *is defined as the* complement *of* t_i *in* t_j *iff*

$$\exists \mathcal{M}_{comp} : [1, n-k] \rightarrow [1,n] \; (injective) \quad |$$

$$(\; Cod(\mathcal{M}_{sub}) \cap Cod(\mathcal{M}_{comp}) \; = \; \emptyset \quad \wedge \quad \forall h \in [1, n-k] : t_{j-i}^h \equiv_\tau t_j^{\mathcal{M}_{comp}(h)} \;)$$

□

In the following, we denote with $\Gamma()$ the function returning the type of an object (in the following, we call *object* a P^3L variable declared with a given type, according to the P^3L data declaration syntax) or a formal parameter.

1.1.2 Input/Output lists

The constructors of P^3L that we will define in the next section use *Input and Output lists.* These lists are made up of the formal/actual parameters that are passed to and are returned by each constructor. The formal parameters are concatenated by means of the operator ';'. If we call par_i a generic parameters, such as $\Gamma(par_i) \in \mathcal{T}$, a list of parameters l has this form:

$$l \; = \; (par_1 \; ; \; par_2 \; ; \ldots \; ; \; par_n).$$

Notice that a list of parameters can also be made up of a single parameter:

$$l \; = \; (par_1)$$

Let l_1 and l_2 be two generic lists of parameters such that:

$$l_1 \; = \; (par_1^1 \; ; \; par_2^1 \; ; \ldots \; ; \; par_n^1)$$

$$l_2 \; = \; (par_1^2 \; ; \; par_2^2 \; ; \ldots \; ; \; par_m^2)$$

we define the concatenation of lists as follows:

$$l_1 ; l_2 = (par_1^1 \; ; \ldots ; \; par_n^1 \; ; \; par_1^2 \; ; \; \ldots ; \; par_m^2)$$

The type of an input/output list is defined as follows:

$$\Gamma(l) = \Gamma((par_1 \; ; \ldots ; \; par_n)) = \langle \Gamma(par_1), \ldots, \Gamma(par_n) \rangle$$

The lengths of the various dimensions of an array type may be left unspecified only in declarations of formal parameters. This means that only the relationship of type equality (see Def. 1.1) must hold between two actual parameters to be associated with the same formal parameter.

As regards the parameter passing, we exploit the definitions given before and concerning structure types. In fact, lists of parameters are just considered structures.

1.1.3 How to denote a partial data type in P^3L

In this section we introduce the language terms of the language, presented by means of a C-like sintax. In particular, we are interested in denoting particular and useful cases of data subtypes. The following sintax will be used:

- if $\Gamma(\mathtt{x}) \equiv_\tau integer[n_1][n_2]$ then

- `x[i][]`, whose type is $integer[n_2]$, denotes the i-th row of `x`
- `x[][i]`, whose type is $integer[n_1]$, denotes the i-th column of `x`
- `x[*][]`, whose type is $integer[n_2]$, denotes a generic row of `x`
- `x[][*]`, whose type is $integer[n_1]$, denotes a generic column of `x`

* if $\Gamma(\mathtt{x}) \equiv_\tau \Gamma(\mathtt{x}) \equiv_\tau int[n]$ then
  - `y[i]`, whose type is $integer$, denotes the i-th element of `y`
  - `y[*]`, whose type is $integer$, denotes a generic element of `y`
  - `y[*i] x[*i]` denote corresponding generic elements of vectors `x` and `y`

The variables like `*i` are called **`free variables`**. They are automatically declared in array subranging using the syntax:

```
<free variable> ::= *<variable>
```

they are integer variables, and when two free variables with the same name appear in the same context (i.e. in the same io-list), they always denote the same value.

Slicing

We introduce the syntax by means of which we can denote a slice of a given array. This denotation individualizes and returns another object (of the same type), that can also be assigned to a variable. So, for each dimension of the array, we have to denote the start index, and the number of elements (of that dimension) that will belong to the slice of the array. If some dimensions are not denoted, the start index is *by default* `0`, while the number of elements is the same of the array to be sliced.

The corresponding syntax is:

```
<slice>::= slice(<vector-name>,<init-index>,<length>) |
           slice(<slice>,<init-index>,<length>)
<vector> ::= <host language vector denotation>
<init-index> ::= <integer>
<length> ::= <integer>
```

Suppose that `v` has been declared as an vector of elements having type t_i. With `slice(v,i,j)` we denote the vector obtaining from `v` taking its slice starting at index `i` and ending at index `i+j-1`.

As an example, let

```
#define N 100
int x[N];
```

be a vector declaration, then

```
slice(x,10,50)
```

denotes a vector of type $integer[50]$ such that its elements are $x_{10}, x_{11}, \ldots, x_{58}, x_{59}$.
Thus,

```
int y[] = slice(x,10,50);
```

is the declaration of the vector

```
int y[50];
```

and the initialization of `y`. The values assigned to the elements of `y` are those individualized by `slice(x,10,50)`.
Let

```
#define N 100
int x[N][N][N];
```

an array declaration, then

```
slice(slice(x,10,50),10,50)
```

denotes an array of type $integer[50][50][100]$ such that its elements are

$$x[10+i_1][10+i_2][i_3] \quad where \quad i_1, i_2 \in [0,\ 49] \wedge i_3 \in [0,\ N-1]$$

1.2 Parallel part

1.2.1 General notation for constructors

Each constructor of the P^3L language can call either P^3L procedures (sequential processes) or P^3L parallel constructors. Hence, constructors can be recursively composed, and it is possible to build a calling tree of the P^3L procedures and parallel constructors making up a P^3L program.

To begin the description of the P^3L language, we first discuss the kind of data allowed and the scoping rules. No global data, if not explicitly declared as *shared*, are permitted. So, the scoping of each constructor is bounded to its input/output list of parameters and to the shared data declarations. If some actual parameters within of the input/output lists have the same name of shared data, then they refer to new data items, distinct from the shared data.

As regards formal/actual parameter specification, it can be written according to the following grammar:

```
<iolist> ::= in(<host language formals>)
             out(<host language formals-actuals>)
<inline iolist> ::= in(<host language actuals>)
                    out(<host language formals-actuals>)
```

Both the input/output lists has been subdivided in a part specifing the input data, and another part specifing the output data of the constructor.
The `<iolist>` is the parameter specification of a P^3L constructor declaration. We call *procedural constructors* those constructors that are first declared along with their `<iolist>`, and then are called by naming their user defined `<process-name>`.
The `<inlist iolist>` is instead the parameter specification of *inline constructors*, that can be regarded as a shorthand for declaring before a specific constructors and calling it subsequently. A constructor declared as *inline* must not be associated with an *user specified name*, because an inline constructor cannot be called in another point of the program, distinct from the point where it is used.
The syntax of the actual parameter list in the call of a procedural constructor is the same of `<inline iolist>`.

Now we explain some particularities of the parameter list syntax presented above. The parameters in the input part of an `<inline iolist>` are actuals, while the parameters included in the input part of an `<iolist>`, i.e. in a declaration of a P^3L constructor, are formals.
When a constructor is called as well as it is declared as inline, the parameters contained in the output part of its parameter list can be *formal* as well as *actual*. They may be actual when

- correspond to global data, i.e. shared data, or
- are variables included in the output part of the constructor just before in the calling tree of the program, or
- are variables contained in the input part of the list itself.

However, these parameters may continue to be formal also in constructor calls or in inline constructors. The meaning of providing formal parameters even in this case is related to the need of declaring a memory buffer where the data produced is stored, and from which another constructor can fetch these data. In other words, the concept is similar to declaring of a channel between two Occam processes, and to bounding the use of such a channel to only two processes.

Moreover, also the parameters included in an <iolist> can be actual. This happens for the paramaters included in the output part of the list, only if they have been declared as formal in the input part of the list.
Due to such a mechanism, some variables in input to a constructor can be returned as output without being modified by the constructor. To distinguish the actual and the formal parameters within the output part of a parameter list, we introduce the functions $\Gamma_A()$ and $\Gamma_F()$. The former takes a list of parameters (<host language formals-actuals>), removes the formals and returns a structure type including the types of the actuals. The latter function takes a list of parameters, removes the actuals and returns a structure type including the types of the formals.
It is worth noticing that, given an <iolist>

in, out

the following property must hold

$$\Gamma_A(out) \subseteq_S \Gamma(in)$$

The syntax for the P^3L constructor calls can be stated as follows:

```
<command-list> ::= <command-item> | <command-item>;<command-list>
<command-item> ::= <p3l constructor call> | <inline p3l constructor>
```

Thus, a command-list is a list of constructor calls or inline constructors. The processes can be either sequential (procedures) or explicitly parallel constructors.

The general syntax of a constructor is the following:

```
<constructor> ::= <inline p3l constructor> | <p3l constructor decl>

<inline p3l constructor> ::= <inline-seq-constr> | <inline-par-constr>
<inline-seq-constr>      ::= <inline iolist> <host-language-body> end
<inline-par-constr>      ::= <constructor-name> <inline iolist>
                             <constructor-body> end <constructor-name>

<p3l constructor decl>   ::= <seq-constr decl> | <par-constr decl>
<seq-constr decl> ::= <process-name> <iolist> <host-language-body> end
<par-constr decl> ::= <constructor-name> <process-name> <iolist>
                         <constructor-body>
                      end <constructor-name>
```

The syntax of a call of a P^3L constructor is the following:

```
<p3l constructor call> ::= <constructor-name> <process-name>
                           <inline iolist>   |
                           <process-name> <inline iolist>
```

The distinction between the inline and procedural classical form of a constructor is no more carried on in the rest of paper. The syntax will be presented for all the constructors and for the procedural case. Inline syntax can be easily derived from the procedural syntax by changing the <iolist> with an <inline iolist> and removing the constructor name.

As far as parameter passing, in P^3L this mechanism is assumed to be *value result*. Data incoming to the constructors are copied to the address space of the constructor, while data outcoming from the constructors are copied to the address space of the caller. In fact, providing formals in the output part of an <inline iolist> may be also seen as a shorthand for the global declaration of such variables in the address space of the caller, and the passing of such variables as *result* parameters to the called constructor.

1.2.2 Constructors

In this part, the parallel constructors are described. Constructors are a small set of forms that can be used to model the parallel structure of the P^3L programs. They are the only forms that can be used to do that. Possibly they can be composed to get more complex parallel structures.

In the constructor list of this first version of P^3L we include:

- pipeline, modeling pipes of processes
- loop, modeling iterative, recursive and history sensitive computations
- sequential process, modeling a single sequential process
- farm, modeling farm, MISD and MIMD parallelism
- tree, modeling static (fixed size) tree parallelism
- reduce, modeling $log - time$ reduce operations
- map, modeling functional apply-to-all parallelism
- geometric, modeling data parallelism

Pipeline

Pipelines are chains of processes where the i-th process sends data only to the (i-th + 1) process and receives data only from the (i-th - 1) process. Some specific parameter passing mechanisms are supplyed for by-passing data through the various processes of the chain.

Pipelines are denoted in P^3L by the keyword `pipe`. In order to use a pipe, the following syntax can be used:

```
<pipe> ::= pipe <process-name> <iolist> <command-list> end pipe
```

Some properties are to hold between the command items making up the `<command-list>`. Let

$$
\begin{array}{l}
\textit{pipe in out} \\
\quad P_1\, in_1\ out_1 \\
\quad\quad cmd_1 \\
\quad \vdots \\
\quad P_n\, in_n\ out_n \\
\quad\quad cmd_n \\
\textit{end pipe}
\end{array}
$$

be the pipe command. Notice that the command items making up the pipe above are *inline constructors*. in denotes the `in` part of the `<iolist>` of the whole pipe, while in_i denotes the `in` part of the `<inline iolist>` of the i^{th} command item (*stage*). The same considerations hold for the `out` parts of the various parameter lists.

The following properties are required to hold (the same properties must hold even if the stages of the pipe were constructor calls):

$$\Gamma(in_1) \subseteq_S \Gamma(in)$$

$$\Gamma_F(out) \subseteq_S \Gamma(out_n)$$

$$(\exists t \in \mathcal{T} \;\mid\; t \subseteq_S \Gamma(out_i) \;\wedge\; t \subseteq_S \Gamma(in_{i+1}) \;\wedge\; \Gamma(in_{i+1}) - \Gamma(t) \;\subseteq_S\; \Gamma(in))$$

The third property states that some information concerning the output part of the i^{th} stage can be forgotten, because not the whole output part of the i^{th} stage is passed to the input part of the stage $i + 1$. This characteristic may be utilized for software reusability, because a module that produces several data items may be utilized even if only a part of such data are necessary.

Moreover, the data in input to a given *pipe* constructor can be by-passed to every other command item simply naming them in the input part of the `<iolist>` of the item itself.

In a pipe command, if $\Gamma(in) \equiv_\tau \Gamma(in_1)$, then *in* can be omitted, as well as if $\Gamma(out) \equiv_\tau \Gamma(out_n)$ then *out* can be omitted.

Loop

The loop constructor models a feedback communication channel between the last and the first enclosed processes.

The syntax for the loop constructor is the following:

```
<loop>::= loop <process-name> <iolist>
                feedback(<actuals-formals>)
                <command-item>
                [<guarded out>]
                until <boolean condition>
          end loop

<guarded out>::= <boolean condition>:out;
```

The `command item` is executed until the `until` boolean condition becomes true. The `guarded out` item can be omitted if the output of the constructor is only provided when the `until` condition is `true`.
The *feedback* parameters are actual if they have been declared as formal in either the input or the output part of the `<iolist>`. Otherwise, also the feedback parameters can be declared as formal. These conditions on the feedback parameters hold for declaration, call and inline form of the loop constructor.

Formally, in a loop constructor such as:

$$\begin{array}{l} loop\ in\ out\ feedback \\ \quad P_b\ in_b\ out_b \\ \quad\quad cmd_b \\ \quad until \\ \quad\quad bool \\ end\ loop \end{array}$$

the following conditions must hold:

$$(\Gamma(in) - \Gamma_A(out)) \subseteq_S \Gamma(in_b) \subseteq_S \Gamma(in)$$

$$\Gamma_F(out; feedback) \subseteq_S \Gamma(out_b)$$

$$\Gamma(feedback) \equiv_\tau \Gamma(in)$$

As in the pipe constructor, the loop in-list (out-list) may be omitted iff $\Gamma(in) \equiv_\tau \Gamma(in_b)$ $(\Gamma(out) \equiv_\tau \Gamma(out_b))$.

Procedure

A procedure is, within P^3L, the simplest way to name a piece of sequential code.

The syntax for the procedure constructor is the following:

```
<procedure> ::= <iolist> <host-language-body> end
```

Farm

Three kind of farms have been included in this version of P^3L:

1. pure farm (workers all perform the same computation onto different data)
2. MISD farm (workers all perform different functions onto the same data)
3. dedicated farm (workers all perform different functions onto different data)

Pure farm The syntax for this farm is the following:

```
<farm> ::= farm <process-name> <iolist> <command-item> end farm
```

Thus, the typical pure farm will have the following structure:

$$\begin{array}{l} farm\ in\ out \\ \qquad P_w\, in_w\ out_w \\ \qquad\quad cmd_w \\ end\ farm \end{array}$$

The following condition on farm data types must hold:

$$((\Gamma(in) - \Gamma_A(out)) \subseteq_S \Gamma(in_w) \subseteq_S \Gamma(in)) \quad \vee$$
$$\vee\ (\exists t \in \mathcal{T} \ \ | \ \ (\Gamma(in) - \Gamma_A(out)) \subseteq_S \ t \subseteq_S \Gamma(in) \quad \wedge \quad \Gamma(in_b) \sharp_S\ t)$$

$$(\Gamma_F(out) \subseteq_S \Gamma(out_w) \Gamma(out)) \quad \vee$$

$$\vee \quad (\exists t \in \mathcal{T} \ \ | \ \ \Gamma_F(out) \subseteq_S \ t \subseteq_S \Gamma(out) \quad \wedge \quad \Gamma(out_w) \sharp_S\ t)$$

In the pure farm, farm in-list (out-list) can be omitted iff $\Gamma(in) \equiv_\tau \Gamma(in_w)$ ($\Gamma(out) \equiv_\tau \Gamma(out_w)$).

MISD farm The syntax of the MISD farm is the following:

```
<farm> ::= farm <process-name> <iolist> <command-list> end farm
```

A typical MISD farm will look like:

$$\begin{array}{l} farm\ in\ out \\ \qquad P_1\, in_1\ out_1 \\ \qquad\quad cmd_1 \\ \qquad \vdots \\ \qquad P_n\, in_n\ out_n \\ \qquad\quad cmd_n \\ end\ farm \end{array}$$

and the following conditions must hold:

$$\forall i \in [1,n] \ \ | \ \ (\Gamma(in) - \Gamma_A(out)) \subseteq_S \Gamma(in_i) \subseteq_S \Gamma(in)$$

$$\exists t \in \mathcal{T} \ \ | \ \ (\ \Gamma_F(out) \subseteq_S \ t \subseteq_S \Gamma(out) \ \ \wedge$$
$$\wedge \ \ (\forall i \in [1,n] \ \ \Gamma(out_i) \sharp_S\ t \quad \vee \quad \Gamma(out_1\ ; \ldots ;\ out_n) \equiv_\tau \ t)\ )$$

Because these two conditions must always hold in the MISD farm, in the general case, the *in* and *out* list can be omitted.

Dedicated farm The syntax of the dedicated farm is the following:

```
<farm> ::= farm <process-name> <iolist> <guard-cmd-list> end farm
<guard-cmd-list> ::= <guard-cmd-item> |
      <guard-cmd-item> <guard-cmd-list>
<guard-cmd-item> ::= <boolean condition> <command-item>
```

A typical dedicated farm will look like:

$$
\begin{array}{l}
farm\ in\ out \\
\quad G_1 : P_1\ in_1\ out_1 \\
\quad\quad cmd_1 \\
\quad \vdots \\
\quad G_n : P_n\ in_n\ out_n \\
\quad\quad cmd_n \\
end\ farm
\end{array}
$$

and the following conditions must hold:

$$\forall i \in [1, n]\ (\ (\Gamma(in) - \Gamma_A(out)) \subseteq_S \Gamma(in_i) \subseteq_S \Gamma(in)\ )$$

$$\forall i \in [1, n]\ (\ \Gamma_F(out) \subseteq_S \Gamma(out_i) \subseteq_S \Gamma(out)\ )$$

Tree

Tree constructor models static tree parallelism, i.e. trees of processes with a predefined, constant number of nodes.

A tree is specified by the following grammar:

```
<tree>::= tree <process-name> <iolist> <mode> <degree-number>
     <root> [<node>] <leaf> end tree
<mode> ::= combining | distributing
<degree-number>::= binary | degree <number>
<root> ::= root <command-item> end root
<node> ::= node <command-item> end node
<leaf> ::= leaf <command-item> end leaf
```

In `distributing` trees, data and control flow from root to leaves, while in `combining` trees, data and control flow from leaves to root nodes.

`root` denotes the tree root process as well as `node` denotes internal (with respect to the tree) processes and `leaf` denotes the leaf processes.

In the `distributing` tree constructor, the root and node processes must have a single item in the in-list and `<degree-number>` items in the out-list. The leaf nodes have just one item in the in-list and one item in the out-list. In the `combining` tree constructor, the root and node processes must have a single item in the out-list and `<degree-number>` items in the in-list. The leaf nodes have just one item in the in-list and one item in the out-list, as in the `distributing` case. The typical structure of a tree is the following:

$$
\begin{array}{l}
tree\ in\ out\ mode\ deg \\
\quad root\ in_r\ out_r \\
\quad\quad cmd_r \\
\quad end\ root \\
\quad node\, in_n\ out_n \\
\quad\quad cmd_n \\
\quad end\ node \\
\quad leaf\ in_l\ out_l \\
\quad\quad cmd_l \\
\quad end\ leaf \\
end\ tree
\end{array}
$$

The **node** part can be omitted whenever the node process performs the same task of the root process.

In the tree constructor, the level number is not specified. It is derived by the compiling tools looking at the data type of the incoming (outcoming) data to (out of) the combining (distributing) tree. This will be explained later. As an example, if a combining tree of degree d has as input an integer vector of l elements, then the tree level number will be $log_d(l)$.

In a distributing tree, the following conditions must hold:

$$\Gamma(in_r) \equiv_{\mathcal{T}} \Gamma(in)$$

$$\Gamma(out_l) \sharp_S \Gamma(out)$$

$$(\exists n \in \mathcal{N}^+ \quad | \quad (\Gamma(out_r) \equiv_{\mathcal{T}} \Gamma(out_n) \equiv_{\mathcal{T}} \Gamma(in_n)[n] \equiv_{\mathcal{T}} \Gamma(in_l)[n]))$$

In a combining tree, the following conditions must hold:

$$\Gamma(in_l) \sharp_S \Gamma(in)$$

$$\Gamma(out_r) \equiv_{\mathcal{T}} \Gamma(out)$$

$$(\exists n \in \mathcal{N}^+ \quad | \quad (\Gamma(in_r) \equiv_{\mathcal{T}} \Gamma(in_n) \equiv_{\mathcal{T}} \Gamma(out_n)[n] \equiv_{\mathcal{T}} \Gamma(out_l)[n]))$$

In both cases, $n \in \mathcal{N}^+$ is equal to 2 iff <degree number> = binary, and $n = <$ number $>$ iff <degree number> = degree <number>.

Reduce

The syntax of the reduce constructor is:

```
<reduce>::= reduce <process-name> <iolist> <operator> end reduce
<operator>::= sum | times | max | min | ... | <binary-proc-call>
```

The typical structure of a reduce constructor will be the following:

reduce in out
    op
end reduce

and the following condition must be satisfied:

$$\exists t \in \mathcal{T} \quad | \quad (\ op : t \times t \rightarrow t \ \wedge \ (\exists n \in \mathcal{N}^+ \mid \Gamma(in) \equiv_{\mathcal{T}} \langle t[n] \rangle) \wedge \Gamma(out) \equiv_{\mathcal{T}} \langle t \rangle \ )$$

The operator op is computed among pairs of composing items of the input parameter first, then among pairs of results, until the root result have been computed.

For instance:

```
reduce in(int x[]) out(int z) sum  end reduce
```

computes the sum of all the elements of vector **x**.

In the reduce operator, either a predefined operation (like **sum** or **times**) is specified, or a user-defined operator. In the latter case, the binary operator is the <process-name> of a sequential constructor having two input parameters and one output parameter, all of the same type $\Gamma(out)$.

The predefined operators **op** must be considered overloaded over the type set $\{integer, real\}$.

The reduce constructor is to be considered a special case of the tree constructor. However, it has been included into P^3L due to the large number of application of reduce operators.

Map

Map is a special case of the pure farm constructor. Its syntax is the following:

```
<map> ::= map <process-name> <iolist> <command-item> end map
```

The typical form of a map constructor will be the following:

$$
\begin{array}{l}
\textit{map in out} \\
\quad P_b \; in_b \; out_b \\
\quad\quad op_b \\
\textit{end map}
\end{array}
$$

and the following condition must hold:

$$\Gamma(in_b)\sharp_S\Gamma(in)$$

$$\Gamma(out_b)\sharp_S\Gamma(out)$$

The *in* and *out* list can be omitted if and only if

$$\Gamma(in_b) \prec_w \Gamma(in)\nabla\Gamma(out_b) \prec_w \Gamma(out)$$

As it happened for the reduce, map constructor has been included in P^3L due to its large number of applications.

Geometric

Geometric constructor is defined in terms of the process and data topology that the programmer wants to implement. Its syntax is the following:

```
<geometric>::= geometric <process-name> <iolist> <topology>
      <process-topology> [<data-topology>] end geometric
<topology> ::= topology <top> [dimensions] [<non-local-comm>]
<top> ::= vector | array | mesh
<non-local-comm> ::= star | row | col | rowcol
<process-topology>::= <process-item> | <process-item> <process-topology>
<process-item> ::= process <indexes> <command-item>
<data-topology>::= <data-item> | <data-item> <data-topology>
<data-item> ::= data <indexes> <actuals>
```

`indexes` is a topology dependent information that allows a single node to be identified uniquely. `dimensions` is a topology dependent parameter. In general it it not required to be specified as it can be statically defined starting from the input list and data topology information. The following indexes and dimensions can be used for typical topologies:

topology	*indexes*	*dimensions*
vector	element-number	n
array	row-number,column-number	n×m
mesh	row-number,column-number	n×m
Mayfly	i,j,k coordinates	s×n

`<non-local-comm>` individualizes the forms of non local communications allowed, i.e. those communications that do not involve neighboring processes w.r.t. the specific topology. Not all of the `<non-local-comm>` parameters works for all the possible topologies. For instance, `star` and `rowcol` only work for `array` and `mesh` topologies.

The typical geometric constructor looks like the following:

$$
\begin{array}{l}
\textit{geometric in out} \\
\quad \textit{topology top dim nonlocal} \\
\quad \textit{process} \ \ i_1 : P_1 \quad in_1\ out_1 \\
\qquad\qquad\qquad cmd_1 \\
\qquad\qquad \ldots \\
\quad \textit{process} \ \ i_k : P_k \quad in_k\ out_k \\
\qquad\qquad\qquad cmd_k \\
\quad \textit{data} \qquad i_1 \ \ data_1 \\
\qquad\qquad \ldots \\
\quad \textit{data} \qquad i_k \ \ data_k \\
\textit{end geometric}
\end{array}
$$

The in-list of each worker P_i specifies the data items that can be read by the worker code, as well as the out-list specifies the data items that can be written. The `data` keyword is optional and specifies the data items that have to be allocated on the node where the corresponding worker process has been mapped. Data items appearing in the worker out-list must also appear in the (implicit or explicit) data list. In fact workers can only write to the data allocated onto their node.

Let us define a new predicate and a function that will be useful to discuss geometric constructor properties.

Definition 1.13 *$t \in \mathcal{T}$ is homomorphic to a topology* `top` *(and we write $t \simeq$ `top`) iff one of the following conditions hold:*

$$(\exists t_b \in \mathcal{T},\ n \in \mathcal{N}^+ \ \mid\ t = t_b[n]) \ \wedge \ (\texttt{top} = \texttt{vector})$$

$$(\exists t_b \in \mathcal{T},\ \ n, m \in \mathcal{N}^+ \ \mid\ t = t_b[n][m]) \ \wedge \ (\texttt{top} = \texttt{array} \vee \texttt{top} = \texttt{mesh})$$

□

Thus, informally, a P^3L data type is *homomorphic* to a topology if it has the same "geometric" structure.

Definition 1.14 *Given a topology* t*, a* `<non-local-comm>` *communication structure, and an index* i *corresponding to a node of that topology, we define* Ω(t,i) *as the set of indexes that denote* neighboring *elements of* i *w.r.t. the topology* t *and the non local communication structure allowed. In other works,* Ω(t,i) *also regards as neighboring two nodes that can communicate each other due to the non local communication structure.* □

Formally, the following conditions must be satisfied by the geometric constructor (with topology t):

$$\Gamma(data_i) \equiv_\tau \Gamma(data_{ii}; data_{io}) \ \ \wedge \ \ (\Gamma(data_{ii}) \sharp_S \Gamma(in)) \ \ \wedge \ \ (\Gamma(data_{io}) \sharp_S \Gamma_F(out))$$

$$\Gamma(in_i) \subseteq_S \Gamma(data_{ii}; data_{j_1}; \ldots; data_{j_k}) \quad \textit{where}\ \{j_1, \ldots, j_k\} = \Omega(t, i)$$

$$(\Gamma(data_{io}) \subseteq_S \Gamma(out_i)) \ \ \vee \ \ (\Gamma(data_{io}) \equiv_\tau \Gamma(out_i))$$

If the *in* list is homomorphic to the constructor topology ($\Gamma(in) \simeq top$), then `data` partition can be omitted. If the *out* list is homomorphic to the constructor topology ($\Gamma(out) \simeq top$), also the `out`-list declaration of each worker can be omitted.

1.2.3 Scoping of parameters

In a P^3L constructor, formal parameters are declared either within the `<iolist>` or within the `feedback` parameters (only for *loop* constructors).
A parameter declared as formal within the input part (output part) of the `<iolist>` can be referred to as actual within the corresponding output and feedback parts (feedback part).
The parameters declared as formal within the input part (output part) of the `<iolist>` can be referred to only within the input part (output part) of the parameter list of the called constructor.

Some further considerations must be made about the pipe constructor, which can call several other constructors. Suppose `int x` is declared within the input part of the `<iolist>` of a pipe constructor. It can be referred within all the input parts of the parameters lists of the various stages of the pipe simply denoting `x`. Furthermore, all the variables declared as formal within the output part of the `<iolist>` of the pipe can be only named within the output part of the parameter list of the last stage.
Furthermore, in a pipe constructor the scoping of a variabled either declared or named within the output part of the parameter list of a given stage is to be considered extended to the input part of the next stage.

As an example, consider the following constructor:

```
loop
  in(int x,t) out(t, int y) feedback(y, t)  // loop iolist
    pipe in(x,t) out(y,t)
      in(x) out(int z)              // pipe stage 1 iolist
      {
      ...
      }
      in(z,t) out(y)                  // pipe stage 2 iolist
      {
      ...
      }
    end pipe
  until (...)
end loop
```

Notice the variable `y`, which is declared as formal in the output part of the loop `<iolist>`. It is then used in the `feedback` part of the `<iolist>` of the loop constructor, in the output part of the `<iolist>` of the pipe constructor, and in the output part of the `<iolist>` of the last stage.

The variable declared as formal parameter in the output part of the `<iolist>` of the first pipe stage (`int z`), is used in the input part of the `<iolist>` of the second (and last) pipe stage.

1.3 Static and shared data

We extend the `command-item` definition as follows:

```
<command-item> ::= <p3l constructor call> | <inline p3l constructor>
                   <sd-command-item>
<sd-command-item> ::= { <sdd> <command-list> }
<sdd> ::= <host language data declaration>
```

where the `sdd` part follows the syntax rules of the host language declarations with the restriction that only P^3L data types can be used (those defined in Section 1.1.1.

We will use data declared this way to model both *shared* data structures and *static* data structures. The former are data structures that can be accessed from a number of different processes. The latter are data structures that can be accessed from just one process, but hold data values across different activations of that process.

1.3.1 Shared data

Shared data can be declared with an `sd-command-item`. In order to declare a shared data structure, it is required that the `command-item` is not a leaf of the constructor tree, i.e. it contains at least some activation of other processes. In term of syntax, this means that the `command-item` must contain at least another constructor.

The data declared in the sdd item will be shared among all the processes belonging to the scope declaration.

Two ways can be used to declared shared data:

- normal (host language) data declarations at the beginning of a block. All the constructors and processes belonging to the block can access the data structure declared this way. As an example, in the code:

  ```
  {
  int a[N][N];

  farm in(...) out(...)
    worker-call( .... )
  end farm
  }
  ```

  the array `a` is accessible from all the workers of the enclosed farm.

- read-only data declarations at the beginning of a block. All the processes enclosed in the block can access the data but none of them can modify the data structure. The keyword `readonly` can be used to declare such a data structure.

1.3.2 Static data

Persistent data, i.e. data structure that survive, with their own data values, across different activation of a given process, can be declared, within P^3L using a `sd-command-item`. If the enclosed `command-item` is a leaf of the constructor tree, i.e. it is a simple process or a simple constructor, then the data declared in the `sdd` part are persistent, i.e. they are allocated at the first activation of the P^3L program and maintain their values until the P^3L program end.

Data declared in such a `sdd` item *must* be initialized.

As an example, a command item can be written now as follows:

```
{
int loop_count=0;

loop
  in(...) out(...) feedback(...)
  {
  ...
  loop_count++;
  ...
  }
 until (...)
end loop
```

declares an integer variable `loop_count` that is allocated (and initialized) when the overall P^3L program is started and rests in memory until the overall P^3L program terminates.

1.4 P^3L programs

A complete P^3L program is made up of a declaration section, and a main section. Declaration section contains all the new data types (`typedef`) constants declaration (`define`) and procedure, function and constructor declarations. Main section contains the program `main` function.

The syntax for a complete P^3L program is the following:

```
<p3l program> ::= <declaration section> <main section>
<declaration section> ::= <a decl> | <a decl> <declaration section>
<a decl> ::= <host language declaration> |
             <p3l constructor declaration list>
<main section> ::= main(<main args>) {<command-list>}
<main args> ::= <any parameter external to p3l,
such as files argv[] etc.>
```

Chapter 2

P^3L operational semantics

In this chapter, an informal, operational semantics for P^3L is discussed.

Semantics for the sequential part of the language is not discussed. The reader should refer to the host language reference manuals for all the stuff concerning the sequential part, but the parameter passing conventions. These conventions are covered by the contents of this chapter.

2.1 Informal semantics

Here the operational semantics of the constructors is described informally. Constructor syntax is that described in Section 1.2.1. The process structure generated by the constructors is that described in Section 2.2.

2.1.1 Pipeline

Pipelines compute functions on an input data stream, whose type is equal to the type of the pipe in-list and produces a stream of results whose type is equal to the type of the pipe out-list.

As soon as an input data is available onto the pipeline input channel, it is passed to the pipeline first stage. When this stage has completed the computation, it sends the output data to the second pipeline stage. At this time, the first stage can start computing a new input data incoming along the pipe input channel.

All the pipeline stages work this way: they wait for data input, compute a result and send it (possibly along with some of the input data) to the next pipeline stage.

The output of the overall pipeline constructor is the data output by the last process.

As the stage P_i of a pipeline can read input data either from the stage P_{i-1} or from the pipe input, then routing from the pipe input channel to the P_i input channel is provided by the P^3L run time support (through stages $P_0, \ldots, P_{i-1}$).

2.1.2 Loop

A loop constructor waits for data to be received along the input channel. Then the data is passed to the enclosed process P_b. The enclosed process computes on the input data, then the boolean conditions are evaluated:

- if the `until` boolean condition is true, the current out-list is returned as result of the computation of the loop constructor and the loop constructor execution is terminated;
- when the `until` condition is false,
  - if the `out` boolean condition is true, the current out-list is passed to the P_e process that outputs it, and the command-item is executed again with the new feedback values as the new input data,

– if the `out` condition is false, the command-item is executed again without outputting anything to the P_e process.

2.1.3 Farm

In the farm constructor, the emitter waits for the input data, then dispatches these data to one of the worker processes. The scheduling can be decided in a self scheduled way, i.e. the idle workers ask a new task, or in a centralized way, by the P_e process, with some kind of fair scheduling policy (round robin, etc.).

The choice on the scheduling policies is P^3L compiler task.

Once the data have been passed to a worker, it starts computing and delivers a result to the collector process P_c. At this point a new task can be scheduled on that worker. The collector process can pass out the result received, if the type of the result matches the farm out-list type, or it can keep it to reconstruct a larger data structure if the worker out-list is a proper subtype of the farm out-list type. While workers compute tasks, the emitter process continues its dispatching activity over the input data.

2.1.4 MISD farm

The MISD farm behaves like a pure farm except that:

- the farm input data is broadcastœ to all the workers allocated within the farm
- each worker performs a different computation onto the input data
- the collector process builds a structure (or a vector) with the results of all the workers

2.1.5 Dedicated farm

In the dedicated farm constructor, the emitter process waits for an input data to appear onto the input channel. Then the boolean conditions associated with the workers are evaluated. The input data is passed to the the worker(s) whose condition is satisfied. The worker compute the results and deliver them to the collector process. The collector process simply outputs the results.

2.1.6 Tree

In a combining tree, once the input data has come to the tree leaves, the leaf processes start computing and producing results according to the `leaf code` semantics. Then, the internal nodes start computing over the results produced by the leaves. Finally the root node computes the output of the tree constructor.

In a distributing tree, the process is reversed. The root starts computing onto the tree input data, then the nodes work, and finally the leaves produce the tree output data.

2.1.7 Reduce

The reduce works as a combining tree, except that the operations performed by leaf, node and root processes are all the same and must implement a binary, associative operator.

2.1.8 Map

The map constructor works as a pure farm, in that it is just a common usage particular implementation of the pure farm.

2.1.9 Geometric

In the geometric constructor, data incoming along the input channel are distributed through the set of processes according to the semantics of the `data` item in the geometric constructor. P^3L compiling tools devise optimal strategies for load balanced data distribution with respect to the target architecture structure.

Once the data has been distributed among the process network, processes start computing and producing results. Results are collected by the ad-hoc code inserted by the P^3L compiler and routed to the geometric constructor output process. In the general case, such a phase it is not needed, or, at least, it does not imply the routing of the results towards a unique output process. In the general case, the results are directly forwarded towards the next computation stage, i.e. towards the next constructor input processes. As an example, if two stages of a pipeline are made up of `geometric vector` constructors, then the output of the first stage does not need any kind of routing through a unique output process. Instead, it can be directly routed towards the next stage input channels.

In the geometric constructor, it is not supposed that processes work for more iterations in sequence. If such a behaviuor has to be obtained, then a geometric-within-a-loop constructor structure has to be adopted. In that case:

- data are allocated to processes (processors) just at the beginning of the computation of the loop constructor;
- P^3L run time guarantees that all the worker processes that need data from the neighbouring processes receive these data *before* the neighbouring processes start modifying them;
- the completion of all the processes of the geometric constructor is awaited before a new loop iteration is started; this is guaranteed, once more, by the P^3L run time code inserted in the user code by the P^3L compiling tools

2.2 Process structures

Each constructor generates, once compiled, a process graph that can be completely defined specifying the following parameters:

- number of the component processes (N_{proc})
- topology of the processes ($G \equiv (N, A)$). The nodes N correspond to processes, the arcs A correspond to internal communication channels (i.e. channels connecting processes belonging to the set N). Each process belonging to the set N is a data structure containing, among the others, the following information:
  - a unique process name
  - the pointer to the associated code
  - the process input channel type
  - the process output channel type

  Each arc belonging to the set A is a data structure containing:
  - the source (writer) process name
  - the drain (reader) process name
  - the type of the data flowing through the channel
- the input-output channel structure of the constructor. A constructor may have a single input (output) channel as well as multiple input or output channels. The input-output channel structure is defined by the data structure $IO \equiv (I, O)$ in which:

- $I \equiv (\{PN_1, \cdots, PN_i\}, \{ChTy_1, \cdots, ChTy_i\})$ defines which processes (PN_j) have an input channel and the input channel data type ($ChTy_j$).
- $O \equiv (\{PN_1, \cdots, PN_i\}, \{ChTy_1, \cdots, ChTy_i\})$ defines which processes (PN_j) have an output channel and the output channel data type ($ChTy_j$).

Let us give some definitions that will be useful later on and that are related with the constructor process structure.

Definition 2.1 *The* basis *of a constructor c (denoted as $\sqcup(c)$) is the ordered set of processes belonging to the constructor that appear in the constructor I set, i.e. if*

$$I \equiv (\{PN_1, \cdots, PN_i\}, \{ChTy_1, \cdots, ChTy_i\})$$

in the constructor, then

$$\sqcup(c) \equiv \{PN_1, \cdots, PN_i\}$$

$\square$

Definition 2.2 *The* ceil *of a constructor c (denoted as $\sqcap(c)$) is the ordered set of processes belonging to the constructor that appear in the constructor O set, i.e. if*

$$O \equiv (\{PN_1, \cdots, PN_i\}, \{ChTy_1, \cdots, ChTy_i\})$$

in the constructor, then

$$\sqcap(c) \equiv \{PN_1, \cdots, PN_i\}$$

$\square$

In the following paragraphs the process structure of each one of the constructors is defined as well as the scheduling policies that have to be adopted and other relevant details of the constructor operational semantics.

In each paragraph, we will refer the names used in the corresponding paragraphs of Section 1.2.1 to describe "typical construct structure".

2.2.1 Pipeline

A pipeline constructor generates the processes that are described by the following graph:

- if n process stages are present in the pipeline constructor, then n processes are created. Each process has its own name, let's say $P_1, \cdots, P_n$.
- the graph $G \equiv (N, A)$ is such that $N \equiv \{P_1, \cdots, P_n\}$ and $A \equiv \{i \in [1, n-1] \mid (P_i, P_{i+1})\}$
- As in the general case, process P_{i+1} may access data either coming from the pipe input list (in) or from the P_i outlist (out_i), the data appearing in the pipe input list in must be "passed through" the whole pipe process chain. This means that the generic channel $a = (P_i, P_{i+1})$ is such that data flowing through it has the same type of $\Gamma(out_i; in)$;
- to each process P_i it is associated the code cmd_i;
- the set I is such that $I \equiv (\{P_1\}, \{\Gamma(in)\})$
- the set O is such that $O \equiv (\{P_n\}, \{\Gamma(out)\})$

The resulting process structure is depicted in Figure 2.1.

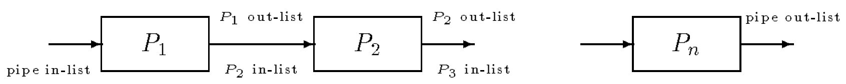

Figure 2.1: Pipeline process structure

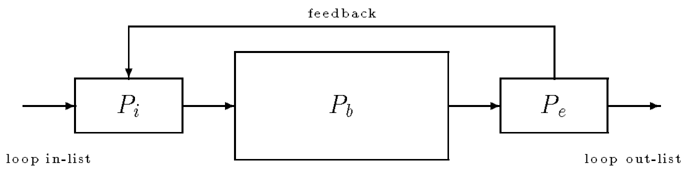

Figure 2.2: Loop process structure

2.2.2 Loop

A loop constructor generates the following process structure:

- two processes are created P_i and P_e
- the graph G is such that

$$N \equiv \{P_i, P_e, P_b\} \ and \ A \equiv \{(P_i, P_b), (P_b, P_e), (P_e, P_i)\}$$

  where P_b is the process relative to the loop body. This process can in turn be either a sequential process or a constructor process graph, with its own input and output process list.

- process P_b (the loop body) has associated the code cmd_b
- process P_i code is generated by the compiler and basically has to provide a fair merge between the data flowing through the loop input channel and that flowing through the feedback channel. Into the process code, code has to be inserted to address the problem of computation identification, if the loop works within a pipeline;
- process P_e code is also generated by the compiler and provides the necessary support for the feedback channel activity;
- I is such that $I \equiv (\{P_i\}, \{\Gamma(in)\})$
- O is such that $O \equiv (\{P_e\}, \{\Gamma(out)\})$

The process structure of the loop constructor is depicted in Figure 2.2.

2.2.3 Farm

The process structure generated by a pure farm constructor is the following:

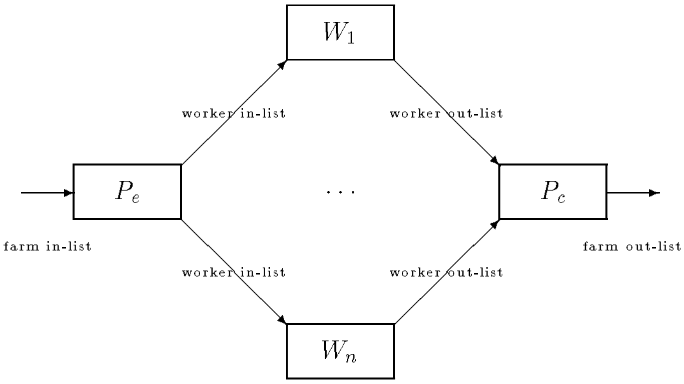

Figure 2.3: Pure farm process structure

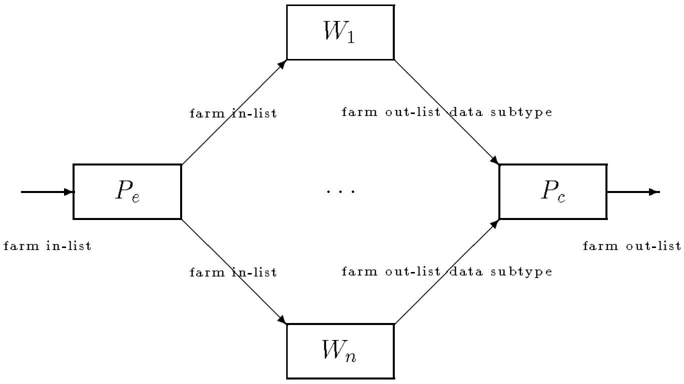

Figure 2.4: MISD farm process structure

- $n+2$ processes are generated, say $P_e, P_c, W_1, \cdots, W_n$. P_e is called the emitter process as well as P_c is called the collector process. The number of worker processes n is decided by the P^3L compiling tools, in such a way that process load is balanced [DPV91a].
- G is such that $N \equiv \{P_e, P_c, W_1, \cdots, W_n\}$ and $A \equiv \{(P_e, W_i), (W_i, P_c) \mid \; i \in [1, n]\}$
- each arc $a \equiv (P_e, W_i)$ has the type $\Gamma(in_w)$
- each arc $a \equiv (W_i, P_c)$ has the type $\Gamma(out_w)$
- $I \equiv (\{P_e\}, \{\Gamma(in)\})$
- $O \equiv (\{P_c\}, \{\Gamma(out)\})$

The process structure of the farm is depicted in Figure 2.3.

2.2.4 MISD farm

The process structure generated by the MISD farm constructor is very similar to the process structure generated by the pure farm constructor except that each arc $a \equiv (P_e, W_i)$ has type $\Gamma(in)$. The process structure of the MISD farm is depicted in Figure 2.4.

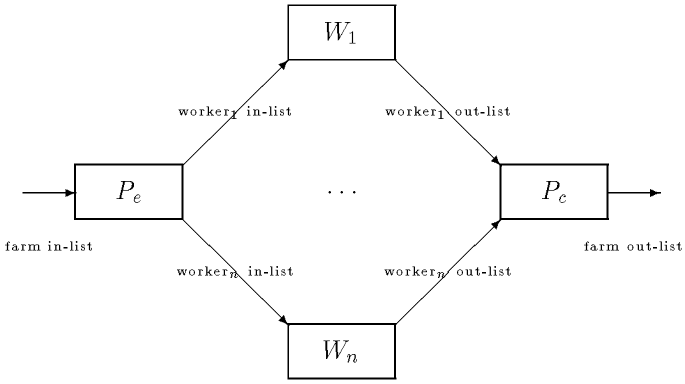

Figure 2.5: Dedicated farm process structure

2.2.5 Dedicated farm

The process structure generated by the dedicated farm constructor is very similar to the process structure generated by the pure farm constructor. The differences are the following:

- each arc $a \equiv (P_e, W_i)$ has type $\Gamma(in_i)$
- each arc $a \equiv (W_i, P_c)$ has type $\Gamma(out)$

The process structure is that depicted in Figure 2.5.

2.2.6 Tree

In order to set up correctly the process structure created by the tree constructor, let us devise some parameters:

N_l N_l is the number of leaf nodes belonging to the tree ($\Delta(in)$ if the tree is `combining`, $\Delta(out)$ otherwise)

d d is the number of sons per node (deg)

N_n $N_n(N_l, d)$ is the number of processes belonging to the tree. This is such that $N_n \equiv \sum_{i=0}^{log_d(N_l)} d^i$.

Node $P_{(l,i)}$ is the $i-th$ node of the $l-th$ level of the tree. Level 0 is the root level, 0-th node at level l is the leftmost one.

Two cases have to be distinguished: `combining` and `distributing` tree.
The process structure for the combining tree is the following:

- $N_n(N_l, d)$ processes belong to the graph
- N is equal to the set of all the $N_n(l, d)$ process-names

$$\{P_{(0,0)}, P_{(1,0)}, P_{(1,1)}, \cdots, P_{(log_d(N_l),0)}, \cdots, P_{(log_d(N_l),N_l-1)}\}$$

- A is such $\forall a \in A \mid a \equiv (P_i, P_j)$ $P_j \equiv P_{(l,h)}$ and $P_i \equiv P_{(l+1,h*d+k)}$ such that $k \in [0, d-1]$ and has type $\Gamma(out_l) \equiv \Gamma(out_n)$
- $I \equiv (\{P_{(log_d(N_l),0)}, \cdots, P_{(log_d(N_l),N_l-1)}\}, \{\Gamma(in_l), \cdots, \Gamma(in_l)\})$
- $O \equiv (\{P_{(0,0)}\}, \{\Gamma(out_r)\})$

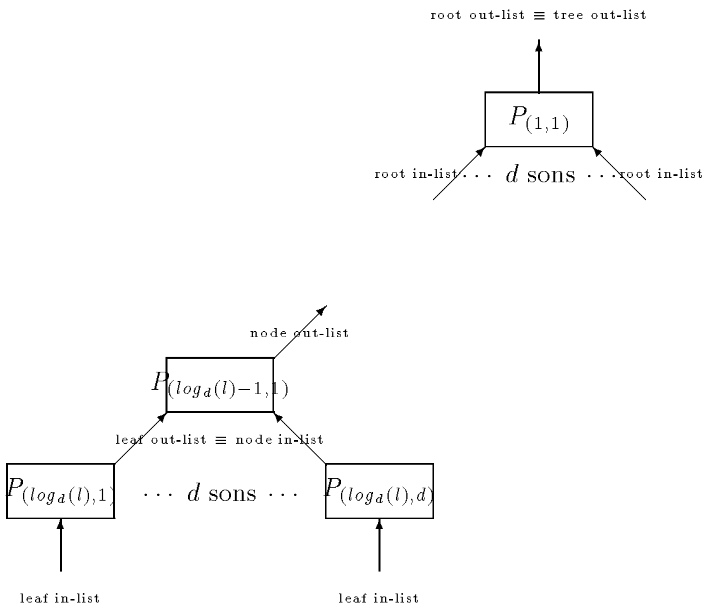

Figure 2.6: Process structure of a combining tree

The process structure of a combining tree is depicted in Figure 2.6.
The process structure of the distributing tree is the same of the combining tree, except that:

- if $a = (P_i, P_j) \in A$ in the combining tree then $a' = (P_j, P_i) \in A$ in the distributing tree (i.e. arcs are reversed) and its type is $\Gamma(in_n) \equiv \Gamma(in_l)$
- $I = (\{P_{(0,0)}\}, \{\Gamma(in_r)\})$
- $O = (\{P_{(log_d(N_l),0)}, \ldots, P_{(log_d(N_l),N_l-1)}\}, \{\Gamma(out_l), \ldots, \Gamma(out_l)\})$

The corresponding process structure is depicted in Figure 2.7.

2.2.7 Reduce

The reduce operator is a special case of the combining tree constructor, having a simple, binary associative operator as unique parameter. The process structure of a reduce such as:

```
reduce in(bbb) out(aaa)
  binop
end reduce
```

will be the same of the combining tree

```
tree combining binary
root binop in(yyy) out(aaa) end root
node binop in(xxx) out(yyy) end node
leaf binop in(bbb) out(xxx) end leaf
```

2.2.8 Map

The process structure induced by the map constructor is the same of that defined by a pure farm constructor.

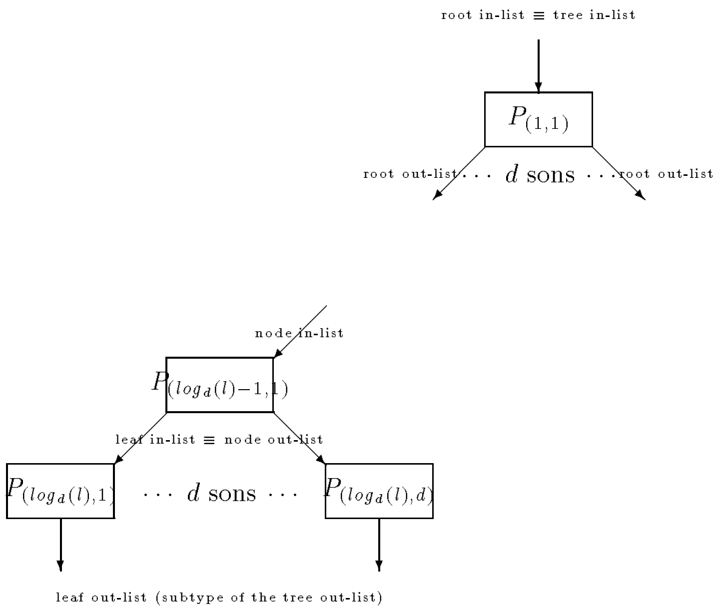

Figure 2.7: Process structure for the distributing tree

2.2.9 Geometric

The process structure of a geometric constructor can be described as follows (depending on the `topology` chosen):

- `vector`:
  - the number of processes belonging to the process graph is n iff n is either the dimension of the vector declared in the `topology` item or it $n = \Delta(in)$;
  - $N \equiv \{P_1, \dots, P_n\}$. Each one of the P_i has associated the code cmd_i;
  - $A \equiv \{(P_i, P_j) \mid (j = (i+1) \vee j = (i-1))\}$
  - $I \equiv (\{P_1, \dots, P_n\}, \{\Gamma(in_1), \dots, \Gamma(in_n\})$
  - $O \equiv (\{P_1, \dots, P_n\}, \{\Gamma(out_1), \dots, \Gamma(out_n\})$
- `mesh,array`:
  - the number of processes belonging to the process graph is either $N_w = n \times m$ if the `topology` item declared a `n,m` dimension or $N_w = \Delta(in)$;
  - $N \equiv \{P_{1,1}, \dots, P_{1,m}, P_{2,1}, \dots, P_{n,m}\}$
  - In the mesh case $A \equiv \{(P_{i,j}, P_{i',j'})\ i \in [0, n-1] \wedge j \in [0, m-1] \mid (i' = (i \pm 1) mod\ n \wedge j' = (j \pm 1) mod\ m)\}$

    In the array case, $A \equiv \{(P_{i,j}, P_{i',j'})\ i \in [1, n-2] \wedge j \in [1, m-2] \mid (i' = (i \pm 1) \wedge j' = (j \pm 1))\}$
  - $I \equiv (\{P_{i,j} \mid i \in [0, n-1] \wedge j \in [0, m-1]\}, \{\Gamma(in_{i,j})\})$
  - $O \equiv (\{P_{i,j} \mid i \in [0, n-1] \wedge j \in [0, m-1]\}, \{\Gamma(out_{i,j})\})$

The process structure of a vector geometric constructor is depicted in Figure 2.8.

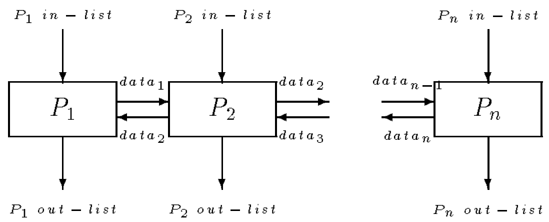

Figure 2.8: Process structure of a vector geometric constructor

2.3 Process mapping

In this section, some problems related with the allocation of constructor component processes are discussed. Each constructor is discussed in a separate paragraph but those constructors that do not present any kind of implications on process-to-processor mapping.

In this section, the following terminology is used to discuss process-to-processor mapping issues:

- two nodes (processors) are *adjacent* if there is a direct link in the Processing Surface that can be used to exchange data between them;
- two processes are *close* if they have to be possibly allocated onto adjacent nodes
- a *direct channel* is a channel that can be implemented using a physical channel between two adjacent nodes
- *plain process* is a process not having other processes as sons

2.3.1 Pipeline

Pipeline stages should be allocated onto distinct, possibly adjacent processors. As stated in section 1.2.1, data flowing in the pipe can be used by the pipe process P_i even if processes P_2 to P_{i-1} does not name these data at all. In this latter case, data can be either passed through all the pipeline stages until P_i is fed, or pipeline process P_1 can be mapped close to P_i in such a way that a direct channel can be used.

In all those cases where pipeline is not used to model sequential processing, the pipeline stages should be allocated onto distinct processors.

2.3.2 Loop

Loop constructors requires non-local communications, in the general case, to implement feedback channels. However, complex mappings can be chosen such that they keep the enclosed constructor input and output processes close enough that the the feedback channel could be implemented as a direct channel.

Some loop processes could also be allocated onto the same processor if just one loop computation is performed at a time. This means that, if no other input data can be accepted while process P_b is computing, then P_i can be merged (placed onto the same node) with the first process of P_b. The same can be done for P_e and P_b last process.

2.3.3 Pure farm

Worker processes should be in general mapped onto different processors. The emitter process could (eventually) be placed onto the same processor of one of the worker processes. This might be done at least in those situations where the emitter process has not too much work to do. The same arguments hold for the collector process.

When farm is enclosed into another constructor, emitter process can be allocated onto the same node of the previous constructor last process. The collector process can be mapped onto the same node of the next constructor first process.

2.3.4 Tree, Map, Reduce

As these constructors represent balanced tree computations, two conditions should be satisfied by mapping:

- processes belonging to the same tree level should be mapped to different processors
- processes having a common father should be mapped onto adjacent nodes

If these conditions hold, then a greater degree of locality can be exploited performing inter-process communications, thus leading to a better performance.

2.3.5 Geometric

The processes of a geometric constructor should be mapped onto different processors. Furthermore, processes that are adjacent with respect to the geometric constructor topology should be mapped onto adjacent processors. This allows the interprocess communications needed to exchange `data` items between processes to be implemented efficiently.

2.4 Process scheduling

The output of the P^3L compiler is a set of processes. These processes have to be run onto the P^3M configuration that represents the compiler target architecture. We've already seen how the processes have to be mapped onto the P^3M configuration. Let's see how they have to be scheduled.

In general, processes are output by the compiler that have to be run onto different processors. A process that must be run alone (i.e. not together with other user processes) onto a particular node, can be either in a `ready` or in a `wait` state. These processes are always created by the compiler as `ready` processes. They are called *standalone processes*. They will wait in this state until their input data are ready (data-flow firing rule). At this point they are moved to the `run` state. They complete their computation and then they are moved back to the `ready state`.

A different situation arises for those processes that must be run onto the same processor (a multiprocessed node). These processes can be in one of three different states: `wait`, `ready` or `run`.

Processes that run in a multiprocessing environment, are created in a `wait` state, become `ready` when input data are ready (data flow scheduling rule) and then become `running` when no other processes with higher priority are `ready`. When they end up their computation they have to be moved back to the `wait` state.

2.4.1 Pure farm

All the processes are scheduled initially into a `ready` state. However, worker processes and emitter-collector processes follow different scheduling rules.

Emitter and collector processes move from the `ready` state to the `run` one as soon as their input data are available. This means that a new input data has arrived at the emitter input channel and that a new result, computed by one of the workers is coming from the collector input channel.

Worker processes are handled according to a self-scheduling policy: a ready list is maintained for worker processes in such a way that, as soon as the emitter wants a new worker, it picks up it from these list. The worker computes, when it ends up, it delivers results to the collector process and puts itself back into the worker ready list.

An alternative strategy can be devised by the emitter process, such as using round-robin policies to handle workers, but self-scheduling achieves best load balancing.

2.4.2 Data flow scheduling

As discussed above, P^3L processes are scheduled according to a data flow firing rule in most cases. In the classical data flow machines, the data flow firing rule is implemented by a dedicate control unit the continuously *searches* for firable instructions in the instruction store. In the P^3L implementation, however, the data flow firing rule is implemented differently. Processes that are scheduled alone on a given node adopt busy waiting. They continuously look for incoming data and start computing once they arrival is complete. Other processes (i.e. those allocated onto multiprocessed nodes) are implemented in such a way that there is no busy waiting. A small P^3L run time kernel is allocated onto such nodes that handles incoming data and user process scheduling.

2.5 Data allocation

Processes produced by the P^3L compiler have different kinds of data that must be correctly allocated before the program is run.

On the one hand, system generated data, such as communication channel data structures, must be allocated somewhere.

On the other hand, user specified data, such as shared data structures or static data associated to a process must also be allocated.

2.5.1 Channel allocation

Channels are used within P^3L to implement inter-process communication. Each channel has its own data structure used to keep track of its status, the rights owned by processes regarding the channel and the buffering data structures.

The data structures used to implement a channel are always associated with the receiving process. This means that they will be allocated in the memory of the processor holding the receiver process.

2.5.2 Static data allocation

Static data must be allocated in the memory of the processor holding the process that works onto these data. Static data are initialized as soon as the program is loaded (actually, they can be generated this way by the compiler). Static data always belong to the address space of the owner process.

2.5.3 Shared data allocation

Shared data is handled in two distinct ways, through P^3L:

- `readonly` data is replicated, when possible, among all the processes that read that data structure;
- read-write data structures are allocated into the P^3M Data Surface.

This has consequences on both P^3L process mapping and scheduling. From the mapping viewpoint, this means that all the processes accessing shared data structures must be adjacent to Data Surface nodes.

From the scheduling viewpoint, as we expect that Data Surface activities (such as retrieving a shared data or storing it) would take much more that Processing Surface activities, this means that processes sharing data must be multiprocessed onto a single node in such a way that mulitprocessing could hide (at least partially) the Data Surface delays.

2.6 Channel linking

P^3L constructors can be recursively composed according to the rules stated in Section 1.2.1. However, from the semantic point of view, some problems arise when two constructors have to be composed which have different number of input and output channels.

The following table recalls the cardinality of input (output) links (i.e. the number of processes linked that wait for (produce) input (output) data):

Constructor	input links	output links
pipeline loop farm map	1	1
tree (combining) reduce	n	1
tree (distributing)	1	n
geometric	n n×n	n n×n

Let's give some definitions concerning constructor I/O structure.

Definition 2.3 *For any constructor c $\iota_\#(c)$ is the cardinality of the first element of the set I as defined in Section 2.2.* □

Definition 2.4 *For any constructor c $\omega_\#(c)$ is the cardinality of the first element of the set O as defined in Section 2.2.* □

Definition 2.5 *For any constructor c such that it has*

$$I = (\{P_{i_1}, \ldots, P_{i_n}\}, \{T_{i_1}, \ldots, T_{i_n}\})$$

$\iota(c, k)$ is defined to be equal to $\Gamma(T_{i_k})$. □

Definition 2.6 *For any constructor c such that it has*

$$O = (\{P_{j_1}, \ldots, P_{j_m}\}, \{T_{j_1}, \ldots, T_{j_m}\})$$

$\omega(c, k)$ is defined to be equal to $\Gamma(T_{j_k})$. □

Definition 2.7 *Two constructors c_1 and c_2* match *(and we write $c_1 \bowtie c_2$) iff*

$$(\iota_\#(c_2) = \omega_\#(c_1) = n) \wedge \forall i \in [1, n](\omega(c_1, i) \asymp \iota(c_2, i))$$

□

Now, no problems arise when constructors c_1 and c_2 have to be pipelined ($c_1 \mid c_2$, borrowing | from unix) and they are such that $c_1 \bowtie c_2$.

If $\neg(c_1 \bowtie c_2)$, then we should be in one of the following cases:

- $\omega_\#(c_1) < \iota_\#(c_2)$
  this means that (recall the table presented above) constructor c_1 has a single output and constructor c_2 has multiple input links. In this case, a process P_d has to be inserted in the composition of c_1 and c_2 such that:
  - it receives data from the last process belonging to constructor c_1 (actually from $P_i \in \sqcap(c_1)$);
  - it broadcasts data to the n input lines of the first processes of constructor c_2 (actually to processes $\{P_j \mid P_j \in \sqcup(c_2)\}$).

If $\omega(c_1, 1) \asymp \iota(c_2, i)[\,]$ for any admissible value of i, then process P_d is a plain router process that routes the first element of the input vector to the first process of the "base" of c_2, the second to the second one, etc.

- $\omega_{\#}(c_1) > \iota_{\#}(c_2)$
  this means that c_1 has multiple output links and c_2 has a single input link. In this case, a process P_d has to be inserted in the composition of c_1 and c_2 such that:
  - it receives data from any of the processes belonging to $\overline{\sqcap}(c_1)$;
  - it delivers data to the single process belonging to $\underline{\sqcup}(c_2)$;
  - if general I/O ordering is required (see Section 2.7), then data coming from lower processes in the basis of c_1 have to be delivered first.

  If $\omega(c_1, i)[\,] \asymp \iota(c_2, 1)$ for any admissible value of i, then process P_d is a plain process that build a vector with data items coming from c_1 and delivers it to c_2.

2.7 Input/Output data ordering

Most of the P^3L processes work within some kind of pipeline. This means that, for instance, the results of the computation of a farm can be used as input for the pipeline next stage, say a geometric constructor.

In the general case, it may be required that the results appear onto the constructor output in the same order of the incoming data that generated these results. More formally, given a constructor $\mathcal{C}$ that computes function $\mathcal{F}$, we require that:

$$\mathcal{C} \quad \ldots, x_j, \ldots, x_{i+k}, \ldots \equiv \ldots, y_{j'}, , \ldots, y_{j'+k}, \ldots$$

$$\forall s \exists t \mid y_s \equiv \mathcal{F}(x_t)$$

$$i < j \rightarrow ((\mathcal{F}(x_i) = y_{i'}, \mathcal{F}(x_j) = y_{j'}) \wedge i' < j')$$

This property is important as in many cases I/O ordering is crucial to the correct program computation. As an example, let us consider a pipeline where:

- first stage is a "user"process that generates requests for a given service A;
- second stage is a "server" process implementing user interface of service A;
- third stage is a "server" process implementing A.

Now suppose that the second stage is a dedicated farm, with one worker process for each one of the requests that the user can issue. Let's also suppose that request execution ordering is essential for the semantics of service A.

In general, the dedicated farm workers will have different delivering times. For instance, time spent by worker W_i could be considerably longer than time spent by worker W_j. In this case, if two requests arrive in sequence, the first one for function i (W_i) and the second one for function j (W_j) then the third pipeline stage will receive the processed request for service A_j and then that relative to A_i.

In order to maintain I/O ordering, some code has to be inserted into the P^3L compiler output. In particular, in the farm constructs, code has to be inserted into the emitter and collector processes in such a way that

- input data is marked with its arrival number (this has to be done in the emitter process);
- worker processes have to be modified to "pass" the input data mark to the collector together with the true result;

- the collector process has to "sort" its input sequence of results with respect to their arrival number

If I/O ordering might turn out to be a necessity, it is also true that it will (potentially) lead to a performance loss in all those cases where it is not required.

This can happen for two distinct reasons:

- because of the unnecessary code inserted for I/O ordering;
- because ordering slows down a pipeline stage, thus slowing down the overall delivering time for results. In other words, it may be the case that computing

  $$F\ x_1, \ldots, x_n \ \equiv F(x_{i_1}), \ldots, F(x_{i_n})$$

  (where x_i represent successive input data to constructor F and $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$) takes less time than computing

  $$F\ x_1, \ldots, x_n \equiv F(x_1), \ldots, F(x_n)$$

2.7.1 Loop task scheduling

A problem similar to that discussed in the previous Section arises with `loop` computations.

If streams of data come to the loop constructor, should we wait for the current loop computation to terminate before starting a new one or not?

As the loop constructor is implemented as a three stage pipeline, some advantage can be obtained from its non-blocking execution. These advantages can either be consistent if the loop body process P_b is a another constructor.

Two solutions can be implemented in P^3L:

- **non-blocking** loops are implemented in such a way that, if new data appears onto the input channel, they are accepted, marker with their arrival order and fed into the loop body input channel. In other words, a mechanism such as that adopted for I/O ordering maintaining is adopted to achieve continuous loop operation;
- **blocking** loops are implemented is such a way that no new input data are accepted until the current loop execution is not terminated.

Both solutions require modifications to the process scheduler. In particular, the former assumes that input and output channels have to be modified to take into account data ordering, and the latter assumes that the data flow firing rule is circumvented.

Chapter 3

Examples

3.1 Toy examples

3.1.1 Pipeline function computation

Suppose that the function:

$$\mathcal{F} : \Re \rightarrow \Re$$

has to be computed. Suppose also that it can be described as the composition of a set functions like follows:

$$\mathcal{F}(x) = f_n(f_{n-1}(\ldots f_1(x))\ldots)$$
$$\forall i \in [1, n]\ f_i : \Re \rightarrow \Re$$

A P^3L program computing this function can be written as:

```
pipe F in(float x) out(float y)
  in(x) out(float x1)
    f1(x,x1)
  in(x1) out(float x2)
    f2(x1,x2)
  ...
  in(xn-1) out(y)
    fn(xn-1,y)
end pipe
```

3.1.2 Zeroes of a function

Given a function $f : \Re \rightarrow \Re$ and an initial "guess" zero, find a true zero.

```
loop
  in(float guess) out(float zero) feedback(zero)

  approx_a_step(guess,zero);

  until (zero < 0.000001)
end loop

approx_a_step(in float g; out float z)
{
// seq code here
// say, for newton method
}
```

3.1.3 Loop with static data

Given a function f we want the following result to be computed:

$$f(f(f(\ldots f(x_0)\ldots)))$$

i.e. f has to be composed N times.

```
{
#define N 100

int n=0;              // static data

loop
  in(float x) out(float res) feedback(res)
  {res=f(x);
   n++;}
  until (n==N)
end loop
}
```

3.1.4 Pure farm

Given a stream of pairs of vectors, compute the stream of their sums, i.e. given a stream of

$$\langle [x_1, \ldots, x_n], [y_1, \ldots, y_n] \rangle$$

as input, compute a stream of

$$[x_1 + y_1, \ldots, x_n + y_n]$$

as output.

```
farm in(int x[],y[]) out(int z[])

  in(x[],y[]) out(z[])
  {
  int i;
  for(i=0;i<N;i++)
    z[i]=x[i]+y[i];
  }

end farm
```

3.1.5 Function server

A set of functions have to be computed over a stream of input data. The function set is $f_1, \cdots, f_n$ and a set $p_1, \cdots, p_n$ has been defined such that if p_i holds on input data, then function f_i is to be computed and the result has to be output from the constructor.

The P^3L program that implements this algorithm is the following (suppose that input and output data are `floats`):

```
farm in(float x) out(float y)
  p1(x) : y=f1(x);
  p2(x) : y=f2(x);
  ...
  pn(x) : y=fn(x);
end farm
```

3.1.6 Fault tolerant computing

The function $\mathcal{F}$ has to be computed in a fault tolerant way. $F_1, \ldots, F_n$ ($n = 2 \times k + 1$) are known algorithms that can be used to compute $\mathcal{F}$.

The following P^3L program computes the "best voted" result of $\mathcal{F}(x_0)$.

```
pipe
  in(int x0) out(int bestFx0)

  farm in(x0) out(int Fx[])
    F1(x0,Fx[0]);
    F2(x0,Fx[2]);

    Fn(x0,Fx[n-1]);
  end farm

  vote(Fx[],bestFx0)
end pipe

vote (in int Fx[]; out int Fx0)
{
// compute the majority result Fx0
...
}
```

3.1.7 Combining tree

Given a vector $x_1, \ldots, x_n$ (n even) compute the following function:

$$f(x) = g(\oplus_{i=1}^{n}(h(x_i)))$$

where $\oplus$ is a binary, associative operator.

```
tree funct_f in(float x[]) out(float res)
  combining binary

  leaf in(x[*i]) out(float lo[*i])
    {lo[*i]=h(x[*i]);}
  end leaf

  node in(float nil, nir)  out(float no)
    {no = nil OPLUS nir;}
  end node

  root in(float ril, rir) out(res)
    {res = g(ril OPLUS rir);}
  end root
```

3.1.8 Vector computation

Given a vector $x_1, \ldots, x_n$, compute the vector $f(x_1), \ldots, f(x_n)$.

```
geometric f_vec in(int x[]) out(int y[])
  topology vector
  data *i: x[*i],y[*i]
  process *i: in(x[*i]) out(y[*i]) {y[*i]=f(x[*i]);}
end geometric
```

that can be simplified as follows:

```
geometric f_vec in(int x[]) out(int y[])
  topology vector
  process *i: {y[*i]=f(x[*i]);}
end geometric
```

3.2 Complete examples

3.2.1 Dot product

The following P^3L program computes the dot product of a stream of vector couples arranged as records.

Thus, if

$$\{[x_1^1, \cdots, x_n^1], [y_1^1, \cdots, y_n^1]\}, \cdots, \{[x_1^m, \cdots, x_n^m], [y_1^m, \cdots, y_n^m]\}$$

is the input to the program, the output will be

$$\sum_{i=1}^{n} x_i^1 \times y_i^1 \cdots \sum_{i=1}^{n} x_i^m \times y_i^m$$

```
pipe dotproduct in(float x[],y[]) out(float z)

  geometric in(x[],y[]) out(float z1[])
    topology vector
    process i : in() out(float z1[*i]) {z1[*i]=x[*i]*y[*i];}
    data i : x[*i],y[*i],z1[*i];
  end geometric

  reduce in(z1[]) out(z)
    sum
  end reduce

end pipe
```

Applying admitted simplifications, the same code can be expressed as:

```
pipe dotproduct in(float x[],y[]) out(float z)

  geometric
    topology vector
    process *i : in() out(float z1[*i]) {z1[*i]=x[*i]*y[*i];}
  end geometric

  reduce
    sum(z1,z)
  end reduce

end pipe
```

3.2.2 Naif matrix multiplication

Given two $n \times n$ matrixes A and B, an $n \times n$ matrix C has to be computed such that:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \times b_{k,j}$$

The naif algorithm for matrix multiplication implementation in P^3L is the following:

```
farm  in(float a[][],b[][]) out(float c[][])
  dotproduct(a[*i][],b[][*j],c[*i][*j]);
end farm
```

3.2.3 Bubble sort

Given a vector $x_1, \ldots, x_n$ a vector $y_1, \ldots, y_n$ such that vector y is a permutation of vector x and $\forall i \in [1, n-1]$ $y_i < y_{i+1}$.

```
geometric exchange in(int x[]) out(int y[])
  topology vector
  process *i: {y[*i]= max(x[*i],x[*i+1]);}
end geometric

loop sort
  in(int x[]) out(int y[]) feedback(int z[])
    exchange(x[],z[]);
  until eq(x[],z[]);
end loop
```

3.2.4 Misra's meeting scheduling

The following program is an implementation of the algorithm proposed by Misra to schedule a meeting.

```
loop in(int time0) out(int themax) feedback(themax)

pipe

  farm alltimes in(time0) out(int nexttime[])
    f1(time,nexttime[1]);
    f2(time,nexttime[2]);
    ...
    fn(time,nexttime[n]);
  end farm

  farm max-eq in(nexttime[]) out(themax,int iseq)
    reduce in(nexttime[]) out(themax) max end reduce
    reduce in(nexttime[]) out(iseq) eq  end reduce
  end farm

end pipe

until (iseq==TRUE)
```

3.2.5 Shortest path

Given a graph $G = (N, A)$ we want to produce an $\#N \times \#N$ matrix `d_min` which d_{ij} entry gives the length of the shortest path from N_i to N_j. The algorithm starts with matrix `d` being the adjacency matrix of graph G, and at each iteration k computes a new matrix in such a way that each element d_{ij} contains the length of the shortest path containing no more than k arcs (if exists) or 0 otherwise.

```
loop shortest-path in(int d[][]) out(int d_min[][]) feedback(d_min[][])

  farm in(d[][]) out(d_min[][])
    find_new_min(d[*i][],d[][*j],d[*i][*j],d_min[*i][*j]);
  end farm

until eq(d[][],d_min[][]);
end loop

pipe find_new_min in(int x[],y[],oldw) out(int neww)

  geometric in(x[],y[]) out(int z[])
    topology vector
    process *i : {z[*i]=x[*i]+y[*i];}
  end geometric

  reduce in(z[]) out(int w)
    min
  end reduce

  in(oldw,w) out(neww)
    {neww=min(oldw,w);}

end pipe
```

3.2.6 Vector convolution

The convolution routine takes two vectors $x_0, \ldots, x_{n-1}$ and $w_0, \ldots, w_{n-1}$, and computes $y_0, \ldots, y_{n-1}$ such that $y_j = \sum_{k=0}^{j} x_k \times w_{j-k}$

```
pipe convolution in(int x[],w[]) out(int y[])

  geometric v_reverse in(w[]) out(int z[])
    topology vector
    process *i : {z[*i] = w[#w-*i];} //let #w be the length of w
  end geometric

  geometric in(x[],z[]) out(y[])
    topology vector
    process *i : dotprod(x[]#(*i),z[]#(*i),y[*i]);
  end geometric

end pipe
```

3.2.7 May's cross

Given a $n \times n$ matrix A, and a function $f()$ such that its type is $f : Int^5 \rightarrow Int$, compute a new matrix B such that $B = B^{(s)}$, $B^{(0)} = A$ and

$$B_{ij}^{(h+1)} = f(B_{ij}^{(h)}, B_{(i-1)j}^{(h)}, B_{i(j-1)}^{(h)}, B_{(i+1)j}^{(h)}, B_{i(j+1)}^{(h)})$$

The corresponding P^3L program is the following:

```
geometric single_cross in(int A[][]) out(int B[][])
```

```
  topology array
  data i,j: A[i][j],B[i][j]
  process i,j:
    in(A[i][j],A[i-1][j],A[i+1][j],A[i][j-1],A[i][j+1])
    out(B[i][j])
    {
    B[i][j]=f(A[i][j],A[i-1][j],A[i+1][j],A[i][j-1],A[i][j+1]);
    }
end geometric

loop cross in(int a[][],s) out(int b[][]) feedback(int b[][],s1)
    pipe
      single_cross(a,b);
      in(s) out(int s1) {s1-=1;}
    end pipe
  until (s==0)
end loop
```

3.2.8 Parallel prefix

Given a vector $[x_1, \ldots, x_n]$, compute the prefix vector

$$[x_1, x_1 + x_2, x_1 + x_2 + x_3, \ldots, x_1 + \ldots + x_n]$$

```
// computes y= y1+x, ... ,yn+x
farm f1 in(int x,y[]) out(int z[])
  {z[*i] = x + y[*i];}
end farm

// given x1, ... ,xn and y1,...,yn
// computes x1,...,xn,xn+y1,...,xn+yn
farm in_node in(int x[],y[]) out(int z[])
  copy(x[],slice(z,1,dim(x))); // copy x to z from x[1] to x[dim(x)]
  f1(x[dim(x)], y, slice(z,dim(x)+1, dim(x)));
end farm

// the prefix tree
tree prefix in(int x[]) out(int px[])
  combining binary

  leaf in(x[*i]) out(int z[])  // leaves build singleton arrays
    {z[0] = x[*i];}
  end leaf

  node in(int nil,nir) out(int no)
    in_node(nil,nir,no);
  end node

end tree
```

Chapter 4

Pragmatics (dell'implementazione)

In this Chapter, we will discuss the problems and open issues that concern the P^3L implementation. Most of the arguments discussed in this Chapter are still discussion topics. Some of the solutions presented have been demonstrated valid, some other are still to be tested. However, the items discussed in this Chapter should make clear which the implementation strategy for P^3L should be and what kind of problems have to be addressed.

4.1 Tool structure

P^3L is implemented through a set of tools that can be referred to as the "P^3L compiler". This is not a traditional compiler, however. Problems such as process to processor mapping, static and dynamic load balancing and speedup optimization have been taken into account. Traditional compiler, even those of explicitly parallel programming languages, do not cope with such problems, usually. Instead, the solution of these problems is demanded to the user that writes the applicative programs.

The structure of the P^3L compiler is depicted in Figure 4.1.

4.1.1 Front-end

The front-end of the P^3L compiler is a classical front-end. It scans source programs, detects language tokens and finally provides an "Abstract syntax representation" of the source code. This representation is then manipulated by the syntax analyzer that produces the internal representation for the program. This representation contains much more explicit information than the original source code.

	Application specification	
(Programmer)	$\downarrow$	
	P^3L source program	
(Front-end)	$\downarrow$	
	Intermediate code	
(Middle-end)	$\downarrow\uparrow$	(P^3M config.)
	Optimized intermediate code	
(Back-end)	$\downarrow$	
	P^3M code	

Figure 4.1: P^3L compiler structure

Abstract syntax representation

Abstract syntax representation will be useful to let the compiler tools operate onto a form which is not disseminated with concrete syntax details.

Most of the first version P^3L compiling tools will be written in Prolog. Thus we thought natural to represent the abstract data syntax of P^3L programs in terms of Prolog facts.

Thus, as an example, the following P^3L segment of code:

```
farm ciccio in(int x[],y[]) out(int z[])
  {
  z[*i] = x[*i] * y[*i];
  }
end farm
```

will be represented by the abstract syntax representation:

```
program_is([
  keyword('farm'),identifier('ciccio'),keyword('in'),
  ...
  keyword('end farm')]).
```

In the abstract syntax representation, functors such as the following must be present:

```
keyword/1, lbracket/0, rbracket/0, ...
identifier/1, separator/0, ...
```

P^3L programs internal representation

The internal representation of the P^3L program is the data structure manipulated by the compiler middle-end.

In general, compiling tools read the internal representation, devise some work, perform it, and produce a new internal representation that substitutes the old one.

As an example, each constructor will be represented in the internal representation with a `constructor` fact. The arguments will be the constructor name, kind, in-list, out-list, shared-data, plus a `process` term for each one of the component processes of the constructor.

The P^3L program segment presented above should be represented internally as:

```
constructor(name(ciccio),
  in_list([var(x,integer(array)),var(y,integer(array))]),
  out_list([var(z,integer(array))]),
  share([]),
  process_list(p1)).

process(name(p1)
  code([assign(subvec(z,i),times(subvec(x,i),subvec(y,i))),return]),
  in_list([x(vector(integer)),y(vector(integer))]),
  out_list([z(vector(integer))]))
```

In general, the following Prolog relations will be present in the internal representation:

- `program`:
  that records the overall structure of the P^3L program that it is going to be compiled; in the `program` relation the following item must be specified (at least):

- constructor tree structure;
- process names
- code pointers

- `constructor`:
  that records all of the details of a given constructor belonging to the P^3L program; in the `constructor` relation the following item must be specified (at least):
  - the name
  - the in-list and out-list
  - the shared data structure used
  - the son process list
  - the parent process list

- `shared_data_structure`:
  that records all the details concerning a particular data structure that is shared among a set of processes; in the `shared data structure` relation, the following items must be indicated (at least):
  - the name of the data structure
  - the dimension of the data structure
  - the type of the data structure
  - the writing process list
  - the reading process list
  - the kind of accesses included in the source program

- `process`:
  that records all the details of a P^3L process. Such process can be the result of the compilation of either a procedure (possibly inline) or a constructor; the following items must be included in the `process` relation:
  - the name
  - the code position
  - the processor allocation
  - the channels used

- `comm_channel`:
  that records all the features of a particular communication channel present in the compiled version of the P^3L program; in the `comm channels` relation the following items must be included:
  - the name
  - the sender process list (possibly with a uniquq process name inside)
  - the receiver process list (possibly with a unique process name inside)
  - the channel type
  - the buffering mode
  - the allocation (I mean, on which processor the channel data structure must be allocated)

P^3M configurations internal representation

P^3L compiling tools will be specialized to compile optimized code for a restricted, although significant, set of computer architectures. The target architectures that we will take into account will be in the MIMD distributed memory class. We will not take into account MIMD shared memory architectures or SIMD architectures. Among all the possible MIMD distributed memory architectures, we will only take into account regular interconnection network architectures such as:

- two dimensional meshes,
- hypercubes,
- exagonal mesh architectures, like the HP Mayfly.

For such architecture, we need a representation that could be exposed to the P^3L compiling tools, in order to make possible optimizations and true parallelizing program transformations.

The model of the physical architecture exposed to the P^3L compiling tools is a P^3M configuration. This configuration needs to be represented at the same level of the internal representation of the P^3L programs.

Thus we assume that the following internal representation of the P^3M configuration will be available to the compiling tools:

- a `topology` Prolog relation, that represents the interconnection topology of the physical machine, along with its dimension;
- a `ps_topology` Prolog relation, that represents the topology of the Processing Surface within the physical machine;
- a `ds_topology` Prolog relation, that represents the topology of the Data Surface within the physical machine;
- a `mechanism_costs` Prolog relation, that represents the (real) costs associated with each one of the mechanisms provided by P^3M.

4.1.2 Middle-end

In the compiler middle-end, all the powerful strategies devised for the P^3L compilation take place. At this level, mapping of constructors to logical process graphs, mapping of logical processes to physical processors, generation of communication channels structure and allocation of data to the Data Surface are performed.

Each one of these tasks (the list is not exhaustive) is very hard. Some of them require the usage of heuristics to be solved, as the general problem is NP-complete. All of them are handled by a part of the P^3L compiler that reads an internal representation of the program, evaluates it and generates a new internal representation. This new representation can either be the old one augmented with some further information, or a new one, possibly (very) different from the old one.

The logical steps belonging to the middle end are implemented through a set of co-routines. These co-routines are called by the P^3L compiler "spine". All of the spine coroutines has an internal program representation as input and an internal program representation as output.

The main co-routines of the compiler spine are:

- Mapping co-routine
- Load balancing co-routine
- Process Graph Optimizer co-routine

Mapping

The mapper tool, i.e. that part of the P^3L middle-end that deals with the process-to-processor mapping, try to devise a good mapping for the process graph generated by the other parts of the P^3L compiler middle-end.

It works using a set of pre-optimized mapping libraries. These libraries provide a set of possible mappings for each one of the constructors (or some of their possible compositions) and for each one of the target architectures that will be taken into account.

All these mappings will be provided along with their costs. The costs will be supplied in a form which is parametric with respect to the costs of the basic mechanism provided by P^3M implementation.

Thus the mapping library will be represented by a Prolog relation such as `mapping` whose arguments will include:

- `mapping` that records the mapping itself;
- `constructor` that records the type of constructor for which the mapping is devised;
- `physical topology` that records the kind of physical architecture for which the mapping is devised;
- `cost` that records the parametric cost of the mapping;
- `dependes_on_hw` that records the architecture dependent parameters that appear in the mapping cost;
- `dependes_on_sw` that records the application dependent parameters that appear in the mapping cost;

Load balancing

Load balancing has also be achieved in a correct P^3L program compilation. In order to load balance a process graph, precise strategies or well defined heuristics can be applied onto the program internal representation.

The static load balancer tool is a co-routine of the compiler spine. It works in phases:

- in the first phase, the current load balancing is measured using a predefined measure function;
- in the second phase, an alternative solution (e.g. an alternative mapping) is devised; this solution may be "suggested" by the compiler spine, in the sense that compiler spine can suggests a strategy to devise a better (with respect to load balancing) solution;
- in the third phase, the alternative solution is evaluated by measuring it, and, the result is passed back to the compiler spine.

The result consists of both a new program internal representation and the evaluation of the speedup (increase of performance) obtained by the new solution with respect to the last one.

At this point the compiler spine must decide if the new solution has to be accepted (the speedup or performance gain may also be negative) or rejected.

Process graph optimizer

This is the third co-routine of the compiler spine. P^3L compiler, once a process graph structure has been devised for the P^3L program, can recognize that some optimizations can be performed. As an example, the collector and emitter of two pipelined farm can be merged. In this case, the process graph optimizer is called. The position where the optimization can be performed is its input parameter as well as the current internal program representation.

The program graph optimizer knows, as libraries, a set of merging (or splitting) optimizations that can be performed onto the process graph.

The program graph optimizer looks at the structure of the program graph at the place indicated by the compiler spine, looks at the optimization library and applies the best optimization possible, if a "positive" optimization exists.

The result is a new program internal representation that is passed back to the compiler spine.

4.1.3 Back-end

The back-end of the compiler is charged to generate the proper code for the P^3M configuration representing the underlying physical architecture.

The P^3L back-end does not generate any kind of assembly language code nor any kind of compiled (architecture specific) code. Instead, the compiler generates a set of relations that can be used to drive P^3M implementation of the source program, plus a set of files containing host language source code. These files must be compiled by the host language (traditional) compiler to obtain the true (native) code that will be eventually run onto the physical architecture.

The output of the P^3L compiler back-end is a set of Prolog relations such as:

- `process_list` that gives the list of Processing surface processes;
- `process_allocation_list` that gives the process-to-processor mapping;
- `processor_scheduling` that gives the scheduling policies adopted for each one of the multiprocesses nodes in the Processing Surface;
- `channel_list` that gives the description of each one of the channels used in the Processing Surface;
- `shared_mem_usage_list` that gives a description of the Data Surface requests issued from the Processing Surface. It also gives the processors that actually access Data Surface;
- `shdata_alloc` that suggest a "close-to-the-usage" allocation of the shared data generated by the program onto the Data Surface nodes;
- `process_code` that give the list of files where the single process code can be found. Such files have been produced by the P^3L compiler in two ways:
  - either picking up parts of the original source code and putting them together to produce parts that are completely host language compliant. Such parts can be compiled by the host language compiler to get source code for the target architecture. It may be the case that P^3L compiler inserts, at the end and at the beginning of each procedure defined in this files, some P^3L specific code, for instance to implement P^3L parameter passing conventions;
  - or generating entirely new code; this happens for instance with the farm emitter/collector processes. This processes are not explicitly indicated in the P^3L source code. They are completely generated by the P^3L compiler. However, the code generated by the compiler is host language compliant also in this case.
- `ds_channel` that gives the complete description of the Processing Surface channels that are implemented through the Data Surface.

It is clear that P^3L compiler back-end must be designed after that the P^3M interface to P^3L has been completely defined (possibly in a definitive way). It is also clear that the P^3L back-end is less critical than the other compiler parts. In fact, most of the information needed to generate the P^3M code will already be present in the internal program representation at the end of the compiler middle-end.

Furthermore, the compiler back-end could also be designed for a set of nude architectures (I mean, without P^3M on top of them) in the case that P^3M design would not have been completed in time.

Chapter 5

Reference

5.1 New Keywords

- `binary degree`
  used to denote the degree of a tree
- `combining distributing`
  denote the type of a tree parallel constructor
- `data`
  denote the data partition assumed in a geometric parallel constructor
- `farm (end farm)`
  enclose the farm parallel constructor
- `geometric (end geometric)`
  enclose the geometric parallel constructor
- `in`
  denotes formal parameters passed by value
- `leaf (end leaf)`
  enclose the leaf code of a parallel constructor
- `loop (end loop)`
  enclose the loop parallel constructor
- `map (end map)`
  enclose the map parallel constructor
- `node (end node)`
  enclose the node code of a tree parallel constructor
- `out`
  denotes formal parameters passed by result
- `pipe (end pipe)`
  enclose the pipeline parallel constructor
- `process`
  denote the process code of a geometric parallel constructor
- `readonly`
  denotes data types that can be accesses only for reading

- `reduce (end reduce)`
  enclose the reduce parallel constructor
- `root (end root)`
  enclose the root code of a tree parallel construct
- `slice`
  denote vector and array slicing
- `sum times max min`
  predefined operators for the reduce parallel operator
- `topology`
  denotes the topology assumed in the geometric parallel constructor
- `tree (end tree)`
  enclose the tree parallel constructor
- `vector array mesh hypercube`
  denote predefined topologies that can be specified in the geometric parallel constructor

5.2 P^3L grammar

Here the overall P^3L grammar is presented.

```
<p3l program> ::= <declaration section> <main section>
<declaration section> ::= <a decl> | <a decl> <declaration section>
<a decl> ::= <host language declaration> | <constructor declaration>
<constructor declaration> ::= <process constructor>
<main section> ::= main(<main args>) {<command-item>}
<main args> ::= <any parameter external to p3l, such as files argv[] etc.>
<readonly decl>::= readonly <host language declaration>
<free variable> ::= *<variable>
<slice>::= slice(<vector-name>,<init-index>,<length>) |
   slice(<slice>,<init-index>,<length>)
<vector> ::= <host language vector denotation>
<init-index> ::= <integer>
<length> ::= <integer>
<inline func> ::= inline <host func declaration>
<proc formals> ::= <formal> | <formal>,<proc formals>
<formal> ::= <io> <host language formal>
<io> ::= in | out
<iolist> ::= in(<host language formals>) out(<host language formals>)
<inline iolist> ::= in(<host language actuals>)
    out(<host language actuals>)
<command-list> ::= <command-item> | <command-item>;<command-list>
<command-item> ::= <command-io> | <process-call>
<command-io> ::= <iolist> <command>
<process-call> ::= <p3l process call> | <host language procedure call>
    | <inline constructor>
<command> ::= <host language command>
<constructor> ::= <inline constructor> | <process constructor>
<inline constructor> ::= <constructor-name> <inline iolist>
      <constructor-body> end <constructor-name>
<process constructor> ::= <constructor-name> <process-name>
      <iolist> <constructor-body> end <constructor-name>
<pipe> ::= pipe [<proc-name>] <command-list> end pipe
<loop>::= loop [<proc-name>]
    in(<formals>) out(<formals>) feedback(<formals>)
    <command-item> [<guarded out>] until <boolean condition>
end loop

<guarded out>::= <boolean condition>:out;
<farm> ::= farm [<proc-name>] <iolist> <command-item> end farm
<farm> ::= farm [<proc-name>] <iolist> <command-list> end farm
<farm> ::= farm [<proc-name>] <iolist> <guard-cmd-list> end farm
<guard-cmd-list> ::= <guard-cmd-item> |
      <guard-cmd-item> <guard-cmd-list>
<guard-cmd-item> ::= <boolean condition> <command-item>
<tree>::= tree [<proc-name>] <iolist> <mode> <degree-number>
     <root> [<node>] <leaf> end tree
<mode> ::= combining | distributing
<degree-number>::= binary | degree <number>
<root> ::= root <command-item> end root
<node> ::= node <command-item> end node
```

```
<leaf> ::= leaf <command-item> end leaf
<reduce>::= reduce [<proc-name>] <iolist> <operator> end reduce
<operator>::= sum | times | max | min | ... | <binary-funct-call>
<map> ::= map [<proc-name>] <iolist> <command-item> end map
<geometric>::= geometric [<proc-name>] <iolist> <topology>
      <process-topology> <data-topology> end geometric
<topology> ::= topology <top> [<dimension>]
<top> ::= vector | array | mesh
<process-topology>::= <process-item> | <process-item> <process-topology>
<process-item> ::= process <indexes> <command-item>
<data-topology>::= <data-item> | <data-item> <data-topology>
<data-item> ::= data <indexes> <actuals>
<command-item> ::= <command-io> | <process-call> |
      <sd-command-item>
<sd-command-item> ::= { <sdd> <command-item> }
<sdd> ::= <host language data declaration>
```

5.3 Typical forms of constructors

5.3.1 Pipe

$$\begin{array}{l}
\textit{pipe in out} \\
\quad P_1\ in_1\ out_1 \\
\qquad cmd_1 \\
\quad \vdots \\
\quad P_n\ in_n\ out_n \\
\qquad cmd_n \\
\textit{end pipe}
\end{array}$$

5.3.2 Loop

$$\begin{array}{l}
loop\ in\ out\ feedback \\
\quad P_b\ in_b\ out_b \\
\qquad cmd_b \\
\quad until \\
\qquad bool \\
end\ loop
\end{array}$$

5.3.3 Pure farm

$$\begin{array}{l}
farm\ in\ out \\
\quad P_w\ in_w\ out_w \\
\qquad cmd_w \\
end\ farm
\end{array}$$

5.3.4 MISD farm

$$\begin{array}{l}
farm\ in\ out \\
\quad P_1\ in_1\ out_1 \\
\qquad cmd_1 \\
\quad \vdots \\
\quad P_n\ in_n\ out_n \\
\qquad cmd_n
\end{array}$$

$end\ farm$

5.3.5 Dedicated farm

$$
\begin{array}{l}
farm\ in\ out \\
\quad G_1 : P_1\ in_1\ out_1 \\
\quad\quad cmd_1 \\
\quad \vdots \\
\quad G_n : P_n\ in_n\ out_n \\
\quad\quad cmd_n \\
end\ farm
\end{array}
$$

5.3.6 Tree

$$
\begin{array}{l}
tree\ in\ out\ mode\ deg \\
\quad root\ in_r\ out_r \\
\quad\quad cmd_r \\
\quad end\ root \\
\quad node in_n\ out_n \\
\quad\quad cmd_n \\
\quad end\ node \\
\quad leaf\ in_l\ out_l \\
\quad\quad cmd_l \\
\quad end\ leaf \\
end\ tree
\end{array}
$$

5.3.7 Reduce

$$
\begin{array}{l}
reduce\ in\ out \\
\quad\quad op\ in_o\ out_o \\
end\ reduce
\end{array}
$$

5.3.8 Geometric

$$
\begin{array}{llll}
\multicolumn{4}{l}{geometric\ in\ out} \\
\quad \multicolumn{3}{l}{topology\ top\ dim} \\
\quad process & i_1 : & P_1\ in_1\ out_1 \\
 & & cmd_1 \\
 & \ldots & \\
\quad process & i_k : & P_k\ in_k\ out_k \\
 & & cmd_k \\
\quad data & i_1 & data_1 \\
 & \ldots & \\
\quad data & i_k & data_k \\
\multicolumn{4}{l}{end\ geometric}
\end{array}
$$

Bibliography

[ABPV88] S. Antonelli, F. Baiardi, S. Pelagatti, and M. Vanneschi. A static approach to process mapping in massively parallel systems. In M. Cosnard, M. H. Burton, and M. Vanneschi, editors, *Parallel Processing*, pages 319–332. IFIP Working Group 10.3, April 1988.

[AG89] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin-Cummings Publishing Company, Inc., 1989.

[AK87] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[ALKK90] A. Agrawal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *The 17th Annual International Symposiun on Computer Architecture*, pages 104–114, May 1990.

[AP90] S. Antonelli and S. Pelagatti. On the complexity of the Mapping Problem for Massively Parallel Architectures. Technical Report TR05/91, Department of Computer Science, University of Pisa (Italy), 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[B+90] S. Borkar et al. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.* ACM-IEEE, June 1990.

[BDJ+91] F. Baiardi, M. Danelutto, M. Jazayeri, S. Pelagatti, and M. Vanneschi. Architectural models and design methodologies for general-purpose highly-parallel computers. In *Proceedings of the IEEE CompEuro '91 – Advanced Computer Technology, Reliable Systems and Applications*, May 1991. Bologna – Italy.

[BDM+91] F. Baiardi, M. Danelutto, R. Di Meglio, M. Jazayeri, M. Mackey, S. Pelagatti, F. Petrini, T. Sullivan, and M. Vanneschi. Pisa parallel processing project on general-purpose highly-parallel computers. In *Proceedings of the COMPSAC '91*, September 1991.

[Bir87] R. S. Bird. An introduction to the Theory of Lists. In Manfred Broy, editor, *Logic of programming and calculi of discrete design.* NATO ASI Series, 1987. International Summer School directed by F. L. Bauer, M. Broy, E. W. Dijkstra and C. A. R. Hoare.

[Bir88] R. S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science.* NATO ASI Series, 1988. International Summer School directed by F. L. Bauer, M. Broy, E. W. Dijkstra and C. A. R. Hoare.

[BJM+91] F. Baiardi, M. Jazayeri, M. Mackey, F. Petrini, and M. Vanneschi. P^3M: an abstract architecture for massively parallel machines. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, April 1991. Leeds, U.K.

[CA90] V. Chaudhary and J. K. Aggarwal. Generalized mapping of parallel algorithms onto parallel architectures. In *International Conference on Parallel Processing*, 1990.

[CE90] S. Chittor and R. Enbody. Performance degradation in large wormhole-routed interprocessor communication networks. In *International Conference on Parallel Processing*, 1990.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation.* Addison-Wesley, 1988.

[DF90] P. K. Dubey and M. J. Flynn. Optimal pipelining. *Journal of Parallel and Distributed Computing*, 8:10–19, January 1990.

[DMPv91a] M. Danelutto, R. Di Meglio, S. Pelagatti, and M. Vanneschi. High level language constructs for massively parallel computing. In *ISCIS VI, The sixth International Symposium on Computer and Information Sciences.* Elsevier, 1991. (to appear).

[DMPV91b] M. Danelutto, R. Di Meglio, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. In *Proceedings of "EIT Workshop on Parallel and Distributed Workstation systems*, September 1991. Florence, Italy (to appear).

[DPV91a] M. Danelutto, S. Pelagatti, and M. Vanneschi. An analytical study of the processor farm. Technical Report HPL–PSC–91-22, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1991.

[DPV91b] M. Danelutto, S. Pelagatti, and M. Vanneschi. High level languages for easy massively parallel programming. Technical Report HPL–PSC–91–16, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1991.

[Dua91] J. Duato. On the design of deadlock-free adaptive routing algorithms for multicomputers: design methodologies. In *PARLE 1991.* LNCS 505, Springer-Verlag, 1991.

[Dun90] R. Duncan. A survey of parallel computer architectures. *IEEE Computer*, 23(2), February 1990.

[DW89] W. J. Dally and D. S. Wills. Universal mechanisms for concurrency. In *PARLE 1989.* LNCS 365, Springer-Verlag, 1989.

[ERL90] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, June 1990.

[FF88] G. C. Fox and W. Furmanski. The physical structure of concurrent problems and concurrent computers. In *Scientific applications of multiprocessors*, pages 55–89. C.A.R. Hoare Series editor, 1988.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salomon, and D. W. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall International, 1988.

[FS90] I. Foster and R. Stevens. Parallel Programming with Algorithmic Motifs. In *International Conference on Parallel Processing*, 1990.

[Gab90] E. Gabber. VMMP: a pratical tool for the development of portable and efficient programs for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), July 1990.

[GG90] C. Germain and J. L. Giavitto. A comparision of two routing strategies for massively parallel computers. In *Proocedings of the Fifth International Symposium on Computer and Information Science*, October 1990. Cappadokia – Turkey.

[Hey88] A. J. G. Hey. Reconfigurable transputer networks: practical concurrent computation. In *Scientific applications of multiprocessors*, pages 39–54. C.A.R. Hoare Series editor, 1988.

[Hey89] A. J. G. Hey. Experiments in MIMD parallelism. In *PARLE 1989*. LNCS 365, Springer-Verlag, 1989.

[KCN90a] C. T. King, W. H. Chou, and L. M. Ni. Pipelined data-parallel algorithms: Part I – Concept and Modeling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4), October 1990.

[KCN90b] C. T. King, W. H. Chou, and L. M. Ni. Pipelined data-parallel algorithms: Part II - Design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4), October 1990.

[KRT88] D. Krizanc, S. Rajasekaran, and T. Tsantilas. Optimal routing algoritms for mesh-connected processor arrays. In J. H. Reif, editor, *VLSI algorithms and architectures*, 1988. Lecture Notes on Computer Science, N.319.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, January 1982.

[Kun88a] M. Kunde. Routing and sorting on mesh connected arrays. In J. H. Reif, editor, *VLSI algorithms and architectures*, 1988. Lecture Notes on Computer Science, N.319.

[Kun88b] H. T. Kung. Computational models for parallel computers. In *Scientific applications of multiprocessors*, pages 1–17. C.A.R. Hoare Series editor, 1988.

[Kun89] H. T. Kung. Computational models for parallel computers. In *Scientific applications of multiprocessors*. Prentice Hall, 1989.

[LRG+90a] V. M. Lo, S. Rajopahdye, S. Gupta, D. Keldsem, A. Mohamed, and J. A. Telle. Mapping divide and conquer algorithms to parallel architectures. In *International Conference on Parallel Processing*, 1990.

[LRG+90b] V. M. Lo, S. Rajopahdye, S. Gupta, D. Keldsem, A. Mohamed, and J. A. Telle. OREGAMI: software tools for mapping parallel computations to parallel architectures. In *International Conference on Parallel Processing*, 1990.

[May88] D. May. The influence of VLSI technology on computer architecture. In *Scientific applications of multiprocessors*, pages 21–37. C.A.R. Hoare Series editor, 1988.

[May90] D. May. Towards general-purpose parallel computers. Draft, also presented at the CRAI Spring International Seminar on Higly Parallel Processing, Capri, May 1990.

[MP90] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1990.

[MS87] D. L. McBurney and M. R. Sleep. Transputer based experiments with ZAPP architecture. In *PARLE 1987*. LNCS 258, Springer-Verlag, 1987.

[NA90] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3), March 1990.

[Pel91] S. Pelagatti. Methodologies and tools for structured highly parallel computing. PhD Thesis Proposal, 1991.

[PJA+87] Pritchard, D. J., Askew, C. R., Carpenter, D. B., A. J. G. Hey, and D. A. Nicole. Pratical parallelism using transputer arrays. In *PARLE 1987*, volume 258 of *Lecture Notes in Computer Science*, pages 28–42. Springer-Verlag, 1987.

[SBKB90] P. A. Suhler, J. Biswas, K. M. Korner, and J. C. Browne. TDFL: a Task-Level Dataflow Language. *Journal of Parallel and Distributed Computing*, 9:103–115, June 1990.

[Ski90a] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, pages 38–50, December 1990.

[Ski90b] D. B. Skillicorn. Practical concurrent programming for parallel machines. Technical Report ISSN-0836-0227-90-282, Department of Computing and Information Science, Queen's University, kingston, Ontario, Canada, July 1990.

[Tse90] P. Tseng. A systolic array parallelizing compiler. *Journal of Parallel and Distributed Computing*, 9:116–127, June 1990.

[Val89a] L. Valiant. Bulk synchronous parallel computers. Technical Report TR-08-89, Aiken Computation Laboratory, Harvard University, 1989.

[Val89b] L. Valiant. General purpose parallel architectures. Technical Report TR-07-89, Aiken Computation Laboratory, Harvard University, 1989.

[WL90] Y. Wu and T. G. Lewis. Parallelizing while loops. In *International Conference on Parallel Processing*, 1990.

[ZRS90] E. Zimran, M. Rao, and Z. Segall. Performance efficient mapping of applications to parallel and distributed architectures. In *International Conference on Parallel Processing*, 1990.

Index