HEWLETT
PACKARD
PISA SCIENCE CENTER
HEWLETT-PACKARD LABORATORIES

HPL–PSC–92–52

November 3, 1992

The P^3L language manual

(current implementation)

Salvatore Orlando

This manual is different from the usual programming language handbooks. In fact, since P^3L is a parallel language that uses an existing sequential language for its sequential part, this manual assumes the knowledge of the sequential part of the language. The current implementation of P^3L uses C++ as the language for expressing its sequential part.

Anyway, P^3L is different from the usual parallel extension of sequential languages. In fact, many of these extensions are libraries including basic mechanisms for communication, synchronization, etc., which must be linked with the sequential part of the language.

Conversely, P^3L gives parallel structure to the sequential language, liberating programmers from low level details like assignment of parallel activities and data, communication and synchronization. The programmer has to declare the kind of parallelism he wants to exploit, and how different forms of parallelism are to be composed for implementing a given parallel application. Distinct P^3L *constructors* can be used for declaring diverse forms of parallelism, and the programmer is requested to express all the parallelism that is present in the application by only using these constructors.

This manual is meant to describe the informal semantics and the syntax of every P^3L constructor, and the way in which these constructors can be composed for writing a parallel application.

1 The *core* P^3L language

The idea of the P^3L language comes from the studies on the structure of many massively parallel applications. These applications have been recognized as using a restricted set of parallel paradigms [Hey89,Kun88], so that, for each of these paradigms, a specific *parallel constructor* has been introduced into the P^3L language. P^3L allows expressing parallel applications by defining such constructors, and composing them in a hierarchical way. Thus, many forms of parallelism can be exploited within the same application.

To introduces the parallel paradigms and the corresponding constructors, it is useful to introduce the following definition of P^3L module.

Each P^3L module $\mathcal{M}$ computes a function on a given stream of input data, and produces an output stream of results. The lengths of both the streams are to be identical, and the ordering has to be preserved, i.e.

$$[in_1, \ldots, in_n] \longrightarrow \mathcal{M} \longrightarrow [out_1, \ldots, out_n]$$

where $[in_1, \ldots, in_n]$ is the input stream, and $[out_1, \ldots, out_n]$ is the output stream. Every output data item out_i is obtained by applying the function computed by $\mathcal{M}$ on the input data item in_i.

The execution model is *data-flow*, i.e. a module waits for an input data item, computes the result, produces an output data item, and so on.

All the data items composing the input stream must be of the same type, and the same property has to hold for all the output data items. Thus, to define a P^3L module, the types

of its input and output interface must be specified, i.e. the types of every in_i and every out_i. These types are lists of parameters with basic and structured types. The dimensions of the types used for describing the module interfaces are to be statically determinable. This restriction is necessary to allow the compiling tools to restructure and optimize the parallel implementations in an effective way.

The P^3L modules can be hierarchically composed. In fact, the function computed by each module can be expressed in terms of other modules. The sequential modules constitute the leaves of the hierarchical composition, because the computation performed by these modules have to be expressed in terms of a sequential language, henceforth called the *host sequential language*. As regards the current implementation of the *core* P^3L, i.e. a restricted P^3L language including the most interesting parallel constructors, the sequential language adopted as host language is C++. C++ was chosen to take advantage of the many tools existing on UNIX for this language, and because of the success of C++ in the industrial environment for the development of large applications.

Until now we have given the definition of a P^3L module. P^3L parallel constructors and sequential processes fall into this definition. In the following, we will use the word *module* to indicate both parallel constructor and sequential processes.

Each parallel constructor corresponds to a form of parallelism belonging to following list:

- **farm** parallelism, i.e. computations where a task dispatcher, called **emitter**, distributes tasks from an input list to a set of "anonymous" **worker** processes. These processes execute the task received, and return the results to a **collector** process. The code to perform the task is given and encapsulated in the worker, while the data on which the task has to be executed is supplied by the emitter. The worker must be another P^3L module;
- **map** parallelism, representing the "easy" *data-parallel* computations. Each input data item is decomposed by the **map_emit** process, and dispatched to a set of "anonymous" **worker** processes. The workers do not need to exchange data to perform their data-parallel computations. The results produced by the workers are returned to a **map_coll** process, which recomposes the output data. The worker must be another P^3L module;
- **pipe** parallelism, i.e. computations conceived as a set of stages to be executed serially over a streams of input data. Each stage must be, in turn, another P^3L module;
- **loop** parallelism, i.e. computations where, for each input data item, a loop body module has to be recursively executed, until a given condition is reached and an output data item is produced. The body of the loop must be another P^3L module.

Fig. 1 shows an example of a P^3L application, in which we can recognize the various P^3L modules, and their hierarchical composition. Notice that, for each module (either sequential process or parallel constructor) the interface is described by means of an *input* (`in(...)`) and an *output list* (`out(...)`). The dots stand for a list of typed parameters.

The corresponding syntax tree, which describes the hierarchical composition of the various P^3L modules is shown in Fig. 2. This tree structure is extensively used by the P^3L compiler for its optimizations.

We can notice that the constructor `pipe main` (the root of the tree) has three nested constructors, i.e. `farm stage1`, `loop stage2`, and `map stage3`.

`pipe main` is also called the *parent* constructor of its nested modules, while the nested modules are also called the *children* of `pipe main`.

2 Data types

The types actually supported by the P^3L level are a subset of those supported by the C++ level.

```
w1 in(...) out(...)
  ${ <seq code> }$
end

w2 in(...) out(...)
  ${ <seq code> }$
end

w3 in(...) out(...)
  ${ <seq code> }$
end

farm stage1 in(...) out(...)
  w1 in(...) out(...)
end farm

loop stage2 in(...) out(...) feedback(...)
  map in(...) out(...)
    w2 in(...) out(...)
  end map

  until <condition>
end loop

map stage3  in(...) out(...)
  farm in(...) out(...)
    w3 in(...) out(...)
  end farm
end map

pipe main in(...) out(...)
  stage1 in(...) out(...)
  stage2 in(...) out(...)
  stage3 in(...) out(...)
end pipe
```

Figure 1: The skeleton of a sample P^3L application

The basic data types supported are the following:
`char`, `boolean`, `int`, `float`.

The previous type can be structured as arrays, denoted as in C++:

```
char var_name[100]
int var_name[10][100]
float var_name[10][10][10], etc.
```

Moreover, it is possible to denote structures composed by sequences of named members of various types. For example:

```
< char a, boolean b[10], float c >
```

where `a`, `b` e `c` are variable names. These structures resemble the `struct` types of C++. It is possible to give a name to this structure by means of the `typedef` declaration:

```
typedef < char a, boolean b[10], float c > STRUCT_NAME
```

The `typedef` declaration can also be used to give different names to basic or structured types. For instance

```
typedef int int_value
```

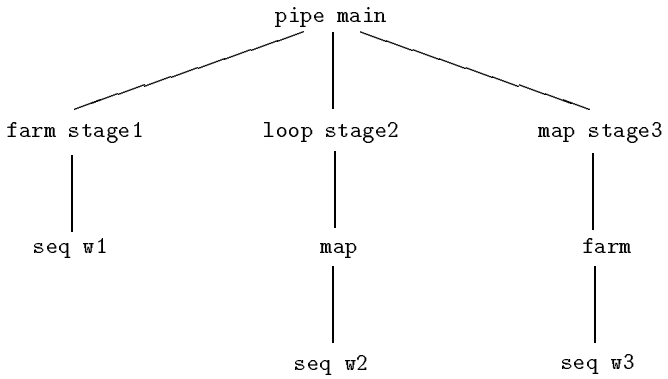

Figure 2: The syntax tree of a sample P^3L application

defines `int_value` as a new denotation of the type `int`;

```
typedef int vect[10]
```

defines `vect` as a new denotation of the vectors composed of 10 integers.

The basic types, the structured ones and the new type denotations introduced by means of the `typedef` definition are used to declare input and output parameters of the P^3L processes and parallel constructors.

In the following we will use the adjectives *formal* and *actual* to syntactically distinguish the following cases, occurring in the input and output parameters lists of P^3L processes and parallel constructors:

- a parameter is "formal" if it appears as in a variable declaration, i.e. the parameter name follows a type denotation;
- a parameter is "actual" if it appears as in a variable reference, i.e. without any associated type.

3 Sequential processes

The *sequential* portions of a P^3L program are written in C++, and are encapsulated into the parallel structures of P^3L. See also [Mac92a].

The input and the output data, specified as lists of parameters having P^3L types, can be considered as a set of variables that can be referenced (as normal C++ variables) by the sequential code.

The following is a very simple example of declaration of sequential process that includes a portion of C++ code:

```
proc in(int a, float b) out(int c)
     ${
        c = a + (int) b;
     }$
end
```

The C++ code has to be included inside two special brackets, i.e. `${` and `}$`.

The name of the process is `proc`. Each input data item elaborated by the process is denoted as a pair of an `int` and a `float` parameter, included in the `in(..)` list. Each output data item produced by the process is denoted as a single `int` parameter, included in the `out(..)` list.

If some *include* files have to be specified, because the definitions contained by these files are used by the code enclosed in the braces, the following syntax must be used:

```
proc in(int a, float b) out(int c)
     inc(f1.H, f2.H)
     $${
        ...
     }$$
end
```

where `f1.H` and `f2.H` are the names of two include files.

C++ procedures, defined inside C++ modules, can also be used to specify the code encapsulated in a P^3L process. Suppose the programmer has defined, in a given `filename.C` file, a C++ procedure `f()`, which takes as formal parameters a list whose members have the same types and the same order as the concatenation of the `in(..)` and the `out(..)` lists. If this is the case, the following syntax can be used:

```
proc in(int a, float b) out(int c)
     f  filename.C
end
```

where `f()` has to be defined as follows:

```
void f(int &var1, float &var2, int &var3)
{
   ...
}
```

We have seen two different ways of declaring the sequential part of a P^3L process. In both the cases, if other C++ modules are to be linked together for generating the final process, the following syntax must be used:

```
proc in(int a, float b) out(int c)
     ...
     src(f1.C, f2.C)
end
```

where `f1.C`, `f2.C` are the other C++ module.

Furthermore, in case some libraries are to be linked to obtain the final process, the following syntax must be used:

```
proc in(int a, float b) out(int c)
     ...
     src(f1.C, f2.C)
     lib(m, codelibs)
end
```

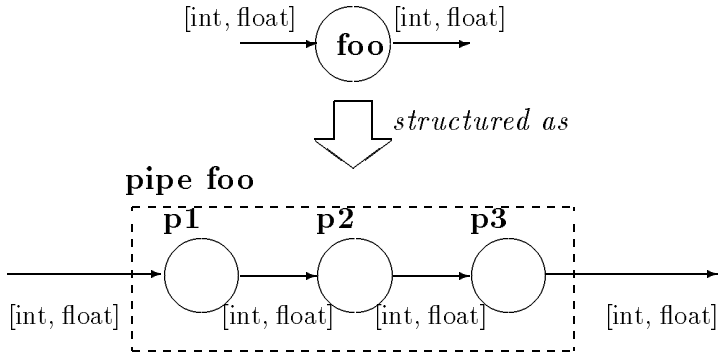

Figure 3: An example of pipe constructor

where `m`, `codelibs` are the names of the libraries.

It is possible to by-pass a parameter from the input list to the output list. Consider the following program:

```
proc in(int a, float b) out(a, int c)
     $${
         c := a + (int) b;
     }$$
end
```

The parameter `int a`, which appears in the input list, is used by the C++ portion of the code, but the same parameter, with the same value as the initial one, has to be returned by the process. The way in which the parameter is passed to the process is similar to "passing by value". In fact, the value of `a` results to be unchanged at the exit of the process, even if it has been updated by the process code. From the syntactical point of view, we have that the same parameter appears in both the input and output lists, but the parameter of the output list is actual.

4 Pipe

The *pipeline* is a paradigm of parallel computation that can be easily expressed by using a P^3L constructor. This paradigm exploits parallelism because of the parallel execution of the various stages composing the pipeline, where each stage is another P^3L module.

In general, each stage starts a new task when the previous one is finished, and the next one arrives. Thus, parallelism can be exploited only if the pipeline is supplied with a stream of data items.

The following piece of P^3L code shows a `pipe` constructor, called `foo`, composed of three stages, the sequential processes `p1`, `p2`, and `p3`. The logical structure of this constructor is shown in Fig. 3.

```
p1 in(int x, float y) out(int z, y)
   $${
        int f1(int);

        z = f1(x);
   }$$
   src(m1.C)
end
```

```
p2 in(int x, float y) out(int z, y)
   ${
        int f2(int);

        z = f2(x);
   }$
   src(m2.C)
end

p3 in(int x, float y) out(int z, float y1)
   ${
        int f3(int);
        float f4(float);

        z = f3(x);
        y1 = f4(y);
   }$
   src(m3.C)
end


pipe foo in(int x, float y) out(int res1, float res2)
     p1 in(x, y) out(int z1, float y1)
     p2 in(z1, y1) out(int z2, float y2)
     p3 in(z2, y2) out(res1, res2)
end pipe
```

The definition of the `pipe foo` constructor follows those of the three sequential stages.

The various stages are called *nested modules*, or also *children* of the *parent* `pipe` constructor (with respect to the the syntax tree of the program).

As shown in the example above, the programmer is requested to write the type of each data item composing the input stream, i.e. the `in(..)` list of the pipe, and the type of each data item composing the output stream, i.e. the `out(..)` list of the pipeline. Moreover, he has to call the nested modules in the right order, taking into account the relations between the types of output and input lists.

Several syntactical hints must be kept in mind to specify a `pipe` constructor, and call its nested modules in the correct way:

- if $\mathtt{p}_i$ is a stage of the `pipe`, and $\mathtt{p}_{i+1}$ is the next stage, then the types of the input list of $\mathtt{p}_{i+1}$ must be identical to the types of the output list of $\mathtt{p}_i$. In other words, the two lists must be of the same length, and corresponding parameters (having the same position) of the two lists must be of the same type.

  Furthermore, if $\mathtt{p}_0$ is the first stage of the `pipe`, then the types of the input list of the whole `pipe` constructor and the types of the at input list of $\mathtt{p}_0$ must be identical. The same property has to hold for the output lists of the whole `pipe` and the output list the last stage $\mathtt{p}_n$ [1];

- the formal parameters of the input list of the first stage must be replaced by the actual ones, i.e. by the name of the parameters of the input lists of the whole `pipe` constructor;

- the formal parameters of the output list of the last stage must be replaced by the actual ones, i.e. the names of the parameters of the output lists of the whole `pipe` constructor;

- the output list of each stage, except the final one, has to be replaced by a list of formal parameters (of the same types as those of the original definition of the constructor). These parameters have to be named as actual parameters in the input list of the next stage.

[1] This is a restriction of the current implementation

The introduction of these new formal parameters can be considered as a definition of a channel between two successive stages. The former stage copies its output data to this channel, while the latter stage reads its input data from the channel.

5 Farm

The *processor farm* is another paradigm of parallel computation that can be easily expressed by using a P^3L constructor.

This paradigm exploits parallelism because of the parallel execution of multiple workers. Each worker computes a distinct task. Since all the workers run using the same code, the programmer is requested to specify only one worker. The worker is, in turn, another P^3L module, also called *nested* module, or *child* of the *parent* `farm` constructor (with respect to the syntax tree of the program).

The informal semantics of the constructor is the following

- a stream of data items, on which the workers have to compute the various tasks, arrives at the *emitter* of the `farm`;
- these data are distributed to the *worker* modules for the computation of the tasks;
- the output of the various workers are collected by the *collector* of the `farm`.

It is worth noticing that the programmer is not request to specify any of the following

- number of workers;
- scheduling of the tasks to the workers.

Both of these are decided by the compiler and by the run-time, on the basis of the costs of the mechanisms exported by the specific parallel system.

The following piece of P^3L code shows a `farm` constructor. The nested module is a sequential processes `p`. The logic structure of this constructor is shown in Fig. 4.

```
p in(float x, int y) out(int z)
   ${
        int f(float, int);

        z = f(x, y);
   }$
   src(m.C)
end


farm foo in(int x, float y) out(int z)
   p in(y, x) out(z)
end farm
```

The definition of the `farm foo` follows the definition of the nested module, i.e. the sequential process `p`.

The definition of the farm includes the call of this nested module. This call expresses the parameter passing from the emitter to each worker (i.e. the nested module), and from each worker to the collector.

As regards the parameter passing between the emitter and the worker, the formal parameters of the input list of the worker are replaced by the actual ones, i.e. the formal parameters of the input list of the `farm`. In this passing, the types of the single actual parameters have to match

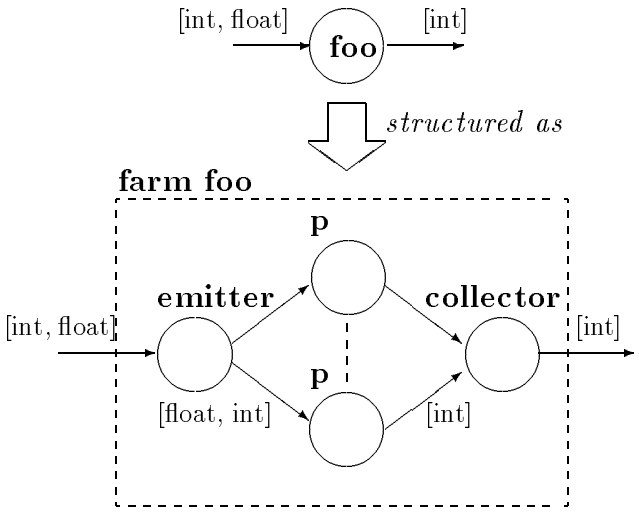

Figure 4: An example of `farm` constructor

the types of the single formals. Thus, it is not necessary that the input lists of both the `farm` and the nested module are identical.

As regards the parameter passing between the worker and the collector, the same rules hold between the output list of the farm and the output list of the nested module. In fact, the formal parameters of the output list of the nested module are replaced by the actual ones, i.e. the formal parameters of the output list of the `farm`.

6 Map

The *data-parallel* paradigm can be expressed by using the P^3L `map` constructor. In this case, parallelism comes from the parallel execution of independent and identical workers, operating on partitions of data items from the input stream. Since all the workers run using the same code, the programmer is requested to specify only one worker. The worker is, in turn, another P^3L module, also called *nested* module, or *child* of the *parent* `map` constructor (with respect to the syntax tree of the program).

The informal semantics of the `map` is the following

- a stream of data items arrives at the *map_emit* process, and a stream of output data items comes from the *map_coll* process;
- each input data item of the stream is partitioned by *map_emit*, and the partitions are distributed to the *workers*. Each worker produces a partition of one output data item of the `map`;
- the outputs of the various workers are received by the *map_coll* of the `map`. This process recomposes the data items to be produced as output of the `map`.

Thus, in order to express a `map` constructor, the programmer is requested to specify the the following

- the nested module, i.e. the worker of the `map` constructor;
- a data decomposition rule, from the types of the input list of the `map` constructor, to the types of input list of the worker;

- a data recomposition rule, from the types of the output list of the worker, to the types of the output list of the `map` constructor.

The data decomposition and recomposition rules are applied to data items structured as arrays. In particular, an array can be decomposed or recomposed taking, for each dimension, either all the subscripts of this dimension, or only one of these subscripts.

As an example, given the array

`int a[10][5]`

the possible denotation of data partitions are the following

`a[*i][]`, `a[][*j]`, `a[*i][*j]`

$*i$ and $*j$ are called *free variables*, and denote one of all the possible subscripts of the dimension in which they appear. As regards the example above, $*\texttt{i} \in \{0, \ldots, 9\}$ and $*\texttt{j} \in \{0, \ldots, 4\}$.

The usage of `[*i]` or `[*j]` to denote a partition means that the partition only includes one of the subscripts corresponding to $*i$ or $*j$. Conversely, the empty brackets `[]` mean, instead, that the partition includes all the subscripts of that dimensions.

It is worth considering the types associated with the denotations of the data partitions. For instance, the following table associates types with array definitions and data partition denotations:

Denotation	*Type*
`int a[10][5]`	int[10][5]
`int b[10]`	int[10]
`a[*i][]`	int[5]
`a[][*j]`	int[10]
`a[*i][*j]`	int
`b[*k]`	int

The programmer has to keep in mind the type associated with each data partition to correctly call the nested module of the `map`.

The following piece of P^3L code shows a `map` constructor. Also in this case the nested module is a sequential process `p`. The logical structure of this constructor is shown in Fig. 5.

```
p in(float x, int y) out(int z)
   ${
        int f(float, int);

        z = f(x, y);
   }$
   src(m.C)
end


map foo in(int x[10], float y[5]) out(int z[10][5])
   p in(y[*i], x[*j]) out(z[*j][*i])
end map
```

The key to understand the decomposition and the recomposition rule, as well as the parameter passing between the `map` and the nested module, is the usage and the meaning of the *free variables*.

For this decomposition of the three arrays (`int x[10]`, `float y[5]`, `int z[10][5]`) two *free variables*, i.e. `*i` and `*j`, have been used. The relative positions of these variables in each partition denotation show the range of these variables to be, e.g. $*\texttt{i} \in \{0, \ldots, 9\}$ and $*\texttt{j} \in \{0, \ldots, 4\}$, and the cardinality of the ranges, e.g. $n_dim(*\texttt{i}) = 10$ and $n_dim(*\texttt{j}) = 5$. The number of data partitions of the input and the output list of the `map` is given by the number of the free variables and by the range of these variables, e.g. $Number_partitions = n_dim(*\texttt{i}) * n_dim(*\texttt{j}) = 10 * 5 = 50$.

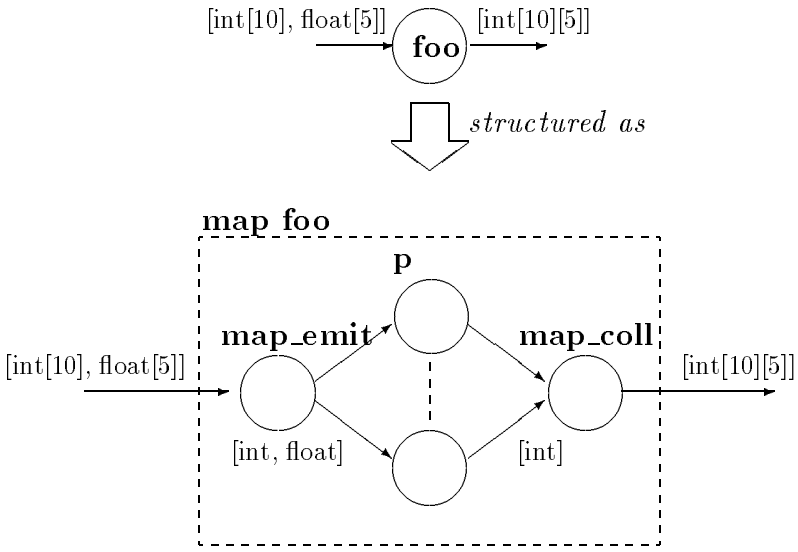

Figure 5: An example of `map` constructor

It is important that, for each data partition of the input of the `map`, there exists one and only one data partition of the output. In other words, a bijective correspondence between the input data partitions and the output ones must be determined. In fact, the elaboration of each input partition has to produce the corresponding output partition, and no output partitions have to be produced by the elaboration of more than one input partition.

The *free variables* determine such a correspondence. The following rules must be followed:

- all the *free variables* that denote the input partitions must be used to denote the partitions of every array structure of the output list of the `map`;
- all the *free variables* that denote output partitions must be used to denote some input partitions, i.e. no further variables can be introduced to denote the output partitions.

Once the usage of the free variables is understood, the parameter passing rules between the input (output) list of the `map` and input (output) list of the nested module are the same as those of the `farm` constructor. It is only necessary to take into account the types associated with each data partition denotation. In fact, the types of the actual parameters (i.e. the data partition denotations) have to match the types of the formal parameters of the nested modules.

7 Loop

The `loop` constructor is used to model *iterative* and *recursive* computations. Parallelism comes from the parallel structure of the nested module, also called *child* of the *parent* `loop` constructor (with respect to the syntax tree of the program).

The informal semantics of the `loop` constructor is the following

- an input stream of data items, each corresponding to a distinct task, arrives at the *loop_in* process of the `loop`;
- the task computation is performed by the nested module. The *loop_in* process sends the data items, each corresponding to a distinct task, to the nested module, which produces an output data item for each input one;

- the output data items arrives at the *loop_out* process of the `loop`; this process decides, on the basis of the value of the data item received, either that a new iteration is needed to compute the task, or that the task is completed and the output data item can be produced.

  In case a new iteration is needed, the data item is sent to the *loop_in* process over the `feedback` channel. Therefore, this process has to perform the merge between the input stream and the feedback stream (coming from the *loop_out* process).

Thus, the parallelism exploited by the nested module is given by the parallel execution of many tasks. In particular, these tasks may come from either the *input data stream* or the *feedback data stream.*

The following piece of P^3L code shows a `loop` constructor. The nested module is a `pipe` composed of three sequential stages.

Each stage of the `pipe` computes a new float expression on taking as parameters the data items received as input, and bypasses the original input data item of the whole `pipe`.

The logical structure of this `loop` constructor is shown in Fig. 6.

```
st1 in(float a) out(float b, a)
  ${
      float f1(float);
      b = f1(a);
  }$
  src(m1.C)
end

st2 in(float a, float old) out(float b, old)
  ${
      float f2(float, float);
      b = f2(a, old);
  }$
  src(m2.C)
end

st3 in(float a, float old) out(float b, old)
  ${
      float f3(float, float);
      b = f3(a, old);
  }$
  src(m3.C)
end

pipe compute_task in(float a) out(float b, float b_old)
  st1 in(a) out(float a1, float old1)
  st2 in(a1, old1) out(float a2, float old2)
  st3 in(a2, old2) out(b, b_old)
end pipe

loop foo in(float a) out(float b)
         feedback(b)
    compute_task in(a) out(b, float b_old)
    until
       in(b, b_old) out(boolean exit_flag)
       ${
           const float threshold = 0.1;

           if ((b_old - b)  >  threshold)
               exit_flag = FALSE;
           else
               exit_flag = TRUE;
```

```
        }$
        end
end loop
```

Looking at the syntax of this `loop` declaration, we can see that, besides the input and the output lists, we also have a `feedback(..)` list of parameters. The types of parameters of `feedback(..)` have to be identical to those of the `in(..)` list of the `loop`. In fact, the `feedback(..)` list contains the data needed for a new iteration of the `loop`. The nested module must be called on the data of either the `in(..)` or the `feedback(..)` list.

The syntax of `in(..)`, `out(..)`, and `feedback(..)` lists describes the parameter passing between the output of the nested module and either the output or the feedback of the `loop`:

- the parameters appearing in the `out(..)` list of the nested module call may be formal or actual:
  1. they are formal when the values of the parameters do not have to be passed either to the `out(..)` nor to the `feedback(..)` list of the `loop`;
  2. they are actual when the values of the parameters have to be passed to the `out(..)` or to the `feedback(..)` list of the `loop`; if this is the case, the same parameters should be named (or declared as formal) in the `out(..)` or in the `feedback(..)` list of the `loop`.
- the parameters appearing in the `out(..)` list of the loop have to be formal, and have to appear as actual in the `out(..)` list of the nested module. This mechanism is used for determining which data items produced by the nested module have to be produced as `out(..)` of the `loop`;
- the parameters appearing in the `feedback(..)` list of the `loop` may be formal or actual. In both the cases, these parameters have to appear as actual in the `out(..)` list of the nested module.

  This mechanism is used to determine which data items produced by the nested module have to be used for the next call of the module itself.

  The parameters appearing in the `feedback(..)` list are actual if they have been already declared as formal in the `out(..)` list of the `loop`. Otherwise, if they are formal, the data item produced by the nested module and used for the new call of the nested module are different from the data item produced as output of the whole `loop`.

In any case, all the data items produced by the nested module must be used to compute the `until` guard. The piece of code that follows the `until` guard is similar to an *inline* call of a sequential process. The `in(..)` list of this process is made up of actual parameters, all coming from the `out(..)` list of the nested module. The `out(..)` list has to include, instead, only one `boolean` parameter (appearing as a formal). The `loop_out` process will use the value of this parameter (which is assigned by the until guard) to decide whether a new loop iteration has to take place, or not. If this parameter results equal to `TRUE`, then the looping stops and the output data are produced. Otherwise, the task is not yet completed, and a new iteration takes place.

As regards the parameter passing between the input list (feedback list) of the loop and the input list of the nested module, the same rules as those for calling the worker of a farm must be followed.

The loop constructor can be called by other P^3L parallel constructors. To call a loop it is only necessary to consider its input and the output parameter lists, but not the feedback list.

Looking at the example above, one of the data items produced by the nested module, i.e. the `pipe compute_task`, is produced as output of the `loop` and as feedback value for the next call of the `pipe`. This data item is `int b`.

The other data item produced by the `pipe`, i.e. `int b_old`, is only used to compute the `until` guard.

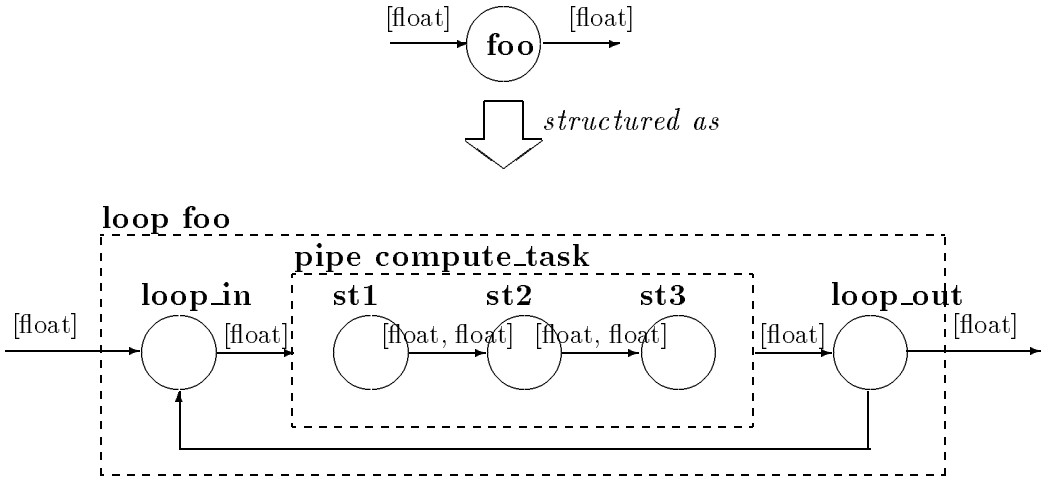

Figure 6: An example of `loop` constructor

8 From `stdin` to `stdout`

The current P^3L compiler prototype allows programmers to write programs that are hierarchically composed of *sequential processes*, *farm*, *map*, *loop*, and *pipe constructors*.

The only constraint regards the outermost constructor, which must be a `pipe`. The first and the last stage of this `pipe` must be special sequential processes. The former has to read the input stream from the `stdin`, while the last stage has to write the output stream to the `stdout`. The programmer has to specify the code of these processes for reading from the `stdin`, and writing to the `stdout`.

The first process has to call a special function `p3l_exit()` when the `EOF` mark is read. The call of this function causes the process to stop reading from the `stdin` and to terminate all the computation.

An example of a process that reads from the `stdin` is the following:

```
init in() out(float a[5], int b)
  ${
       if (fscanf(stdin, "%f", &a[0]) == EOF)
           p3l_exit();
       else
       {
           for (int i=1; i<5; i++)
               fscanf(stdin, "%f", &a[i]);
           fscanf(stdin, "%d", &b);
       }
  }$
end
```

Only this process may have an empty input list, `in()`.

An example of process that writes to the `stdout` is the following:

```
final in(int a[10], float b) out()
  ${
       fprintf(stdout, "a:\t");
       for (int i=0; i<10; i++)
           fprintf(stdout, "%d ", a[i]);
```

```
        fprintf(stdout, "\nb:\t%f\n", b);
  }$
end
```

Only this process may have an empty output list, `out()`.

9 The structure of a P^3L program

In a P^3L program we can distinguish three consecutive parts:

1. the first part in which constants and macros (`#define ..`) are defined, as well as new types (`typedef ..`) are introduced;
2. the second part in which parallel constructors and sequential processes are declared. It is worth noticing that the *scoping* is *static*. Thus, if the definition of a constructor (process) is needed for declaring a new constructor, the declaration of the former constructor (process) has to appear first;
3. the third and main part of the program, that has to be structured as follows:

   ```
   main
     pipe_main in() out()
   end
   ```

   where `pipe_main` is the name of a `pipe` constructor. The first stage of the `pipe` is the process that reads from the `stdin` (its input list is empty). The last stage of the of the `pipe` is the process that writes to the `stdout` (its output list is empty).

10 `inline` constructors

In many cases, since a given process or parallel constructor is only called by a single constructor, an `inline` notation can be used. This notation can be regarded as a shorthand for declaring a specific constructor first, and calling it subsequently.

The following example is the same as that used to introduce the `farm` constructor. In this case, the sequential process (the nested module of the farm) is declared using an `inline` notation:

```
farm foo in(int x, float y) out(int z)
   in(y, x) out(z)
       ${
          int f(float, int);

          z = f(y, x);
       }$
       src(m.C)
   end
end farm
```

Notice that, in this case, no names have been given to the nested process. In fact, this `inline` process cannot be called by other constructors.

Using the `inline` notation, the types of the input/output lists are inferred from the types of the parameters of the parent constructor[2]. For instance, the `inline` sequential process of the example takes a pair of parameters as input. The type of this pair is `(float, int)`, and is

[2] This is partially true, because in case an `inline` module appears as an intermediate stage of a `pipe` constructor, the parameters of the output list must be declared as formals. Thus, the types of the output parameters of this module are given by the types of the formals.

given by the types of the formal parameters that appear in the input list of the `farm` (the parent constructor), and that have been passed to this process.

The `inline` notation can be used to declare any P^3L module[3].

Suppose that the same `farm` as that illustrated above has to be used as a stage of a `pipe` constructor. The following is the declaration of the `pipe`, where the `farm` is now declared using the `inline` notation:

```
pipe foo in(int x, float y) out(int z)
   stage1 in(x, y) out(int x1, float y1)

   farm in(x1, y1) out(int z1)
      in(y1, x1) out(z1)
          ${
             int f(float, int);

             z1 = f(y1, x1);
          }$
          src(m.C)
      end
   end farm

   stage3 in(z1) out(z)
end pipe
```

Notice that, in this case, no names have been provided for the `inline farm`.

[3]Some limitations exist for the `map` constructor. In fact, suppose that the nested module is a sequential process. Since the parameters passed to this process are data partitions, these partitions cannot directly be used in a piece of C++ code, because C++ does not recognize the data partition denotations. Thus, the `inline` notation cannot be used for the nested module of the `map`.

Appendix A: Some examples of P^3L programs

In the following we present some simple P^3L programs. We omit the declaration of the standard processes that performs the reading from the `stdin` (called `start`) and the writing to the `stdout` (called `stop`).

Matrix multiplication

The following is a P^3L program that computes the *matrix multiplication* on a stream of pairs of matrices `float a[5][4]` and `float b[4][6]`. The process `start` and the process `stop` are the first and the last process of the main pipe, respectively.

```
////////////////////////////////////////////////////////////
// Sequential process that performs the scalar multiplication
// of two vectors
////////////////////////////////////////////////////////////
seq_w in(float a[4], float b[4]) out(float c)
  ${
      c = 0;

      for (int i=0; i<4; i++)
          c += a[i] * b[i];
  }$
end

////////////////////////////////////////////////////////////
// Map constructor that applies the scalar multiplication two
// every pair (row, column) of the two matrices a[][] and b[][]
////////////////////////////////////////////////////////////
map w1 in(float a[5][4], float b[4][6]) out(float c[5][6])
    seq_w in(a[*i][], b[][*j]) out(c[*i][*j])
end map



////////////////////////////////////////////////////////////
// Main part of the program
////////////////////////////////////////////////////////////
main
  pipe in() out()
    start in() out(float a[5][4], float b[4][6])
    w1 in(a, b) out(float c[5][6])
    stop in(c) out()
  end pipe
end
```

Shortest path

The *shortest path* problem that we are going to illustrate is concerned with the determination of the shortest paths between each pair of nodes of a completely connected graph. The P^3L program is able to compute the shortest path on a stream of these graphs.

Each graph is represented by a two dimensional matrix. At the beginning, the entry of the matrix corresponding to the row i and the column j represents the distance between the node i and the the node j of the graph. The distance corresponds to the weight of the arc between i and j.

Since there exist many paths that connect i and j, the algorithm looks for the shortest path, i.e. the path having the smallest weight, where the weight of the path is the sum of the weights of each arc composing the path.

The process `start` and the process `stop` are the first and the last process of the main pipe, respectively.

```
////////////////////////////////////////////////////////////
// Sequential process that performs the sum of two integers,
// and bypasses the value of old_min
////////////////////////////////////////////////////////////
ww1 in(int x, int y, int old_min) out(int z, old_min)
   ${
       z = x + y;
   }$
end

////////////////////////////////////////////////////////////
// Map constructor that applies the sum to each pair of
// integers belonging to the two vectors x[] and y[].
////////////////////////////////////////////////////////////
map w1 in(int x[5], int y[5], int old_min) out(int z[5], int old_min1)
    ww1 in(x[*i], y[*i], old_min) out(z[*i], old_min1)
end map



////////////////////////////////////////////////////////////
// Sequential process that determines the minimum value among
// the elements of an integer vector z[] and the integer
// old_min. The value of old_min is bypassed.
////////////////////////////////////////////////////////////
sum_vect in(int z[5], int old_min) out(int min, old_min)
  ${
      min = z[0];

      for (int i=1; i<5; i++)
          if (z[i] < min)
             min = z[i];
      if (min > old_min)
          min = old_min;
  }$
end


////////////////////////////////////////////////////////////
// Pipe constructor that sums two vectors and finds the minimum
// of the obtained vector and the value of old_min.
// The value of old_min is bypassed.
////////////////////////////////////////////////////////////
pipe find_new_min in(int x[5], int y[5], int old_min)
                  out(int new_min, int old_min1)
    w1 in(x, y, old_min) out(int z[5], int old_min_tmp)
    sum_vect in(z, old_min_tmp) out(new_min, old_min1)
end pipe



////////////////////////////////////////////////////////////
// Map constructor that applies the pipe find_new_min to the
// row d[*i][] and the column d[][*j] of the distance matrix d[][].
// This application finds the path *i->k->*j (for each k) that
// has the minimum weight.
// The old distance d[*i][*j] is bypassed to check whether the
```

```
// new distance has been improved w.r.t. the old one.
////////////////////////////////////////////////////////////
map find_min_i_j in(int d[5][5]) out(int d_min[5][5], int d_old[5][5])
    find_new_min in(d[*i][], d[][*j], d[*i][*j])
                 out(d_min[*i][*j], d_old[*i][*j])
end map



////////////////////////////////////////////////////////////
// Loop constructor that applies the map find_min_i_j to the
// matrix d[][]. The new matrix d_min[][] and the old one
// d_old[][] are compared by the until guard of the loop.
// If they are equal, then the fixed point of the algorithm has
// been reached, and the shortest path matrix is produced.
// Otherwise, the map find_min_i_j is re-applied to the new
// matrix d_min[][] (d_min[][] appears in the feedback()
// list of the loop).
////////////////////////////////////////////////////////////
loop shortest_path in(int d[5][5]) out(int d_min[5][5])
                   feedback(d_min)
    find_min_i_j in(d) out(d_min, int d_old[5][5])

    until
      in(d_min, d_old) out(boolean b)
      ${
        b = TRUE;

        for (int i=0; i<5 && b; i++)
            for (int j=0; j<5 && b; j++)
                if (d_min[i][j] != d_old[i][j])
                    b = FALSE;
      }$
      end
end loop



////////////////////////////////////////////////////////////
// Main part of the program
////////////////////////////////////////////////////////////
main
  pipe in() out()
    start in() out(int d[5][5])
    shortest_path in(d) out(int d_min[5][5])
    stop in(d_min) out()
 end pipe
end
```

Appendix B: P^3L grammar[4]

Data types

```
<type-specifier> ::= char | boolean | int | float |
                     <typedef-name>



<typedef-definition> ::= typedef <type-specifier> <typedef-name> |
                         typedef <struct-specifier> <typedef-name>
                         typedef <type-specifier> <typedef-name> <array-dim>


<array-specifier> ::- <type-specifier> <var-identifier> <array-dim>
<array-dim> ::=  [<constant-int-exp>] | <array-dim>[<constant-int-exp>]


<array-partition-specifier> ::= <var-identifier> <array-dim-partition>
<array-dim-partition> ::= <subscript-specifier> |
                           <subscript-specifier> <array-dim-partition>
<subscript-specifier> ::= [*<var-identifier>] | []


<struct-specifier> ::= <  <list-formal>  >


<type-def> ::= <array-specifier> | <type-specifier> <var-identifier>



<list-formal> ::= <type-def> | <type-def>, <list-formal>
<list-formal-actual> ::= <var-identifier> | <type-def> |
                         <var-identifier> , <list-formal-actual> |
                         <type-def>, <list-formal-actual>
<list-actual> ::= <var-identifier> | <var-identifier>, <list-actual>
<list-actual-partition> ::= <var-identifier> | <array-partition-specifier> |
                            <var-identifier>, <list-actual-partition> |
                            <array-partition-specifier>, <list-actual-partition>


<in-list> ::= in(<list-formal>)
<out-list> ::= out(<list-formal-actual>)
<feedback-list> ::= feedback(<list-formal-actual>)

<in-list-actual> ::= in(<list-actual>)
<out-list-formal> ::= out(<list-formal>)
<out-list-actual> ::= out(<list-actual>)

<in-list-actual-partition> ::= in(<list-actual-partition>)
<out-list-actual-partition> ::= out(<list-actual-partition>)
```

[4] Without inline notations

Sequential process

```
<seq-definition> ::= <proc-name> <in-list> <out-list>
                                 <body-seq-definition>

<seq-definition-input> ::= <proc-name> in() <out-list>
                                       <body-seq-definition>

<seq-definition-output> ::= <proc-name> <in-list> out()
                                        <body-seq-definition>



<body-seq-definition> ::= <body-seq> end |
                          <body-seq> <src> end |
                          <body-seq> <libs> end |
                          <body-seq> <src> <libs> end


<body-seq> ::= ${ <C++-code-portion> }$ |
               inc(<list-include-files>) ${ <C++-code-portion> }$ |
               <name-C++-procedure> <name-C++-module>


<src> ::= src(<list-of-files>)
<libs> ::= libs(<list-of-libraries>)
```

Pipe constructor

```
<pipe-definition> ::= pipe <proc-name> <in-list> <out-list>
                           <body-pipe-definition>

<body-pipe-definition> ::= <list-of-calls> end pipe

<list-of-calls>   ::= <first-call>
                      <list-of-calls-inner>
                      <last-call>

<first-call> ::= <proc-name> <in-list-actual> <out-list-formal>

<last-call> ::= <proc-name> <in-list-actual> <out-list-actual>

<list-of-calls-inner> ::= <proc-name> <in-list-actual> <out-list-formal> |
                          <proc-name> <in-list-actual> <out-list-formal>
                          <list-of-calls-inner>
```

Farm constructor

```
<farm-definition> ::= farm <proc-name> <in-list> <out-list>
                           <body-farm-definition>


<body-farm-definition> ::= <call> end farm

<call> ::= <proc-name> <in-list-actual> <out-list-actual>
```

Map constructor

```
<map-definition> ::= map <proc-name> <in-list> <out-list>
                          <body-map-definition>

<body-map-definition> ::= <call-map> end map

<call-map> ::= <proc-name> <in-list-actual-partition>
                          <out-list-actual-partition>
```

Loop constructor

```
<loop-definition> ::= loop <proc-name> <in-list> <out-list> <feedback-list>
                          <body-loop-definition>


<body-loop-definition> ::= <call>
                          until
                             <in-list-actual> <out-list_formal>
                                    <body-seq-definition>
                          end loop
```

The structure of a P^3L program

```
<p3l-program> ::= <prologue-definitions>
                  <p3l-constr-def-list>

                  pipe <name-pipe-main> in() out()
                       <body-pipe-definition>
                  end pipe

                  main
                       <name-pipe-main> in() out()
                  end




<prologue-definitions> ::= <typedef-definition> |
                          <macro-constant-definition> |
                          <typedef-definition> <prologue-definitions>
                          <macro-constant-definition> <prologue-definitions>

<name-pipe-main> ::= <proc-name>

<p3l-constr-def-list> ::= <constr-definition> |
                         <constr-definition> <p3l-constr-def-list>

<constr-definition> ::= <seq-definition> |
                        <seq-definition-input> |
                        <seq-definition-output> |
                        <pipe-definition> |
                        <farm-definition> |
                        <map-definition> |
                        <loop-definition>
```

References

[DDMS+92] M. Danelutto, R. Di Meglio, Orlando S., S. Pelagatti, and Vanneschi M. A Methodology for the Development and the Support of Massively Parallel Programs. *FGCS J.*, 8:205–220, 1992.

[DOP91] M. Danelutto, S. Orlando, and S. Pelagatti. P^3L: The Pisa Parallel Programming Language (Ver. 1.0). Technical Report HPL-PSC-91-27, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1991.

[Hey89] A.J.G. Hey. Experiment in MIMD Parallelism. In *Proc. of Int. Conf. PARLE '89*, pages 28–42, Eindhoven, The Netherlands, June 1989. LNCS 366 Spinger-Verlag.

[Kun88] H.T. Kung. Computational Models for Parallel Computers. In C.A.R. Hoare Series editor, editor, *Scientific applications of multiprocessors*, pages 1–17. Prentice-Hall International, 1988.

[Mac92a] M. Mackey. P^3L interface to C++. Technical Report HPL-PSC-92-50, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.

[Mac92b] M. Mackey. P^3L parser. Technical Report HPL-PSC-92-47, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.