HEWLETT PACKARD
PISA SCIENCE CENTER
HEWLETT-PACKARD LABORATORIES

HPL–PSC–93–55

May 3, 1993

P^3L: a Structured High-level Parallel Language, and its Structured Support

Bruno Bacci[*], *Marco Danelutto*[°], *Salvatore Orlando*[°], *Susanna Pelagatti*[°], *Marco Vanneschi*[°]

(°)Dipartimento di Informatica
Università di Pisa
Corso Italia 40
Pisa – Italy

(*)Pisa Science Center
Hewlett Packard Laboratories
Vicolo del Ruschi, 2
Pisa – Italy

This paper presents a parallel programming methodology that ensures easy programming, efficiency, and portability of programs to different machines belonging to the class of the general-purpose, distributed memory, MIMD architectures. The methodology is based on the definition of a new, high-level, explicitly parallel language, called P^3L, and of a set of static tools that automatically adapt the program features for each target architecture.

P^3L does not require programmers to specify process activations, the actual parallelism degree, scheduling, or interprocess communications, i.e. all those features that need to be adjusted to harness each specific target machine. Parallelism is, on the other hand, expressed in a structured and qualitative way, by hierarchical composition of a restricted set of language constructs, corresponding to those forms of parallelism that are frequently encountered in parallel applications, and that can efficiently be implemented.

The efficient portability of P^3L applications is guaranteed by the compiler along with the novel structure of the support. The compiler automatically adapts the program features for each specific architecture, accessing the costs (in terms of performance) of the low-level mechanisms exported by the architecture itself. In our methodology, these costs, along with other features of the architecture, are viewed through an abstract machine, whose mechanism interface is used by the compiler to produce the final object code.

Contents

1 **Introduction** . . . 3

2 **The P^3L language** . . . 4

3 **The target architecture and its abstract machine** . . . 7

4 **Implementation templates of the P^3L constructs** . . . 10

4.1 Process networks implementing the P^3L constructs . . . 10

4.1.1 The `sequential` construct . . . 11

4.1.2 The `farm` construct . . . 11

4.1.3 The `map` construct . . . 14

4.1.4 The `pipe` construct . . . 15

4.1.5 The `loop` construct . . . 16

4.1.6 Termination . . . 16

4.2 Composition and mapping . . . 17

4.3 The analytical performance models . . . 17

5 **The structure of the compiler** . . . 21

5.1 The libraries . . . 22

5.2 The front-end . . . 25

5.3 The middle-end . . . 25

5.4 The back-end . . . 31

6 **Related work** . . . 32

7 **Conclusions and future work** . . . 33

References . . . 34

1 Introduction

Many parallel architectures exist on the market, and a language (and a corresponding computational model) that can be efficiently ported to different machines is still an subject of research. In particular, a big challenge is the definition of an easily usable and portable language for massively parallel processing.

While easy usability of a parallel language depends on its high-level features, i.e. on the ability to abstract enough from implementation details of the actual target machine, the concept of portability is twofold. It requires not only that a program runs on different machines without any source modifications, but also that the porting is able to exploit the specific architectural features of each machine. In fact, even when two parallel systems are based upon the same architectural model (e.g. they have the same memory model, or the same interconnection network topology), it is known that degree of parallelism, task granularity, data/process mapping have to be adjusted to exploit the specific features of each machine (e.g. number of processing elements, computation/communication ratio, memory size, etc.).

In this paper we present a new programming methodology that ensures both easy programming and efficient portability of programs to different machines belonging to the class of the general-purpose, distributed memory, MIMD architectures (DM-MIMD). The methodology is based on

- a high-level, structured, explicitly parallel language, called P^3L (*Pisa Parallel Programming Language*) [8],
- a set of compiling tools that, exploiting the structured features of P^3L, realize the efficient portability of applications, and
- an abstract machine (AM), called P^3M, which exports all those features of the underlying architecture that the compiling tools need to restructure the implementation of each application.

P^3L is a *high-level* parallel language since programmers can abstract from low-level implementation details, such as process mapping, interprocess communications, process scheduling, and so on, and can concentrate on the forms of parallelism (i.e., paradigms of parallel computation) that must be exploited to parallelize a given application. Some examples of these forms of parallelism are the *pipeline*, the *processor farm*, the *map*, etc. P^3L supplies distinct language constructs for each of these forms of parallelism, and allows them to be hierarchically composed to express more complex forms of parallel computation. The possibility of expressing a given application as a composition of many parallel components, of a manageable size and a definite behavior, allows P^3L to be regarded as a *structured* language.

The choice of the forms of parallelism to include as primitive language constructs in P^3L comes from the experience of programmers of parallel applications. In fact, applications programmed to exploit massive parallelism make use of a *limited number* of forms of parallelism, which exhibit regular structures in terms of both partitioning and replication of functions and data, and of interconnection structures among the processes. Many of them had already been analyzed and presented in [12,13]. These forms of parallelism and their compositions can be implemented on different massively parallel architectures, taking substantial advantages from exploiting locality. Furthermore, each of these possible implementations, henceforth called *implementation templates*, can be analytically modeled, so that simple performance formulae can be associated with them.

The P^3L constructs corresponding to the various forms of parallelism are the *only* means a programmer has to give *parallel structure* to an application. We were comforted in this choice by the evolution of programming languages. In fact, a look at the history of imperative languages shows the progressive disuse of very basic constructs, e.g. `goto`'s, in favor of a set of constructs with definite control flow behavior, e.g. `while`, `if`, `for`, etc. The introduction of these new constructs has not only improved programmer's productivity (software development and maintenance), but has also made a lot of compiling optimizations possible, allowing more efficient implementations to be devised. Thus, we decided to dismiss basic parallel mechanisms

from P^3L, such as `send/receive` or `parallel process activation`, in favor of a reduced set of parallel constructs, each expressing a specific form of parallelism. Using P^3L to design parallel applications, programmers have to declare the *kind* of parallelism they want to use, by *structuring* applications by means of the P^3L parallel constructs and their hierarchical composition. The sequential parts, on the other hand, are expressed by using a sequential language, henceforth called the *host sequential language*.

In our programming methodology, the efficient portability of programs relies on the compiler, which take great advantage of the structured features of the language. In fact, the compiler includes all the knowledge about the *implementation templates* of the various P^3L constructs and their compositions on different DM-MIMD machines (e.g. machines adopting interconnection networks with distinct topologies). In order to effectively exploit the features of each target architecture, the compiler tunes each program implementation using the performance model, and, more specifically, the analytical formulae, associated with the implementation templates of the various P^3L constructs. These formulae are parameterized with respect to the costs of the low-level mechanisms exported by the target architecture. For this purpose, in our methodology, each architecture is viewed through an AM, which exports the interface of the mechanisms to the compiler along with the associated costs.

The aim of this paper is to discuss in more detail the overall design of the compiler, and the implementation templates of the various P^3L constructs. The need to consider diverse architectures as possible targets of P^3L suggested a particular modular structure of the compiler design. In fact, a set of "libraries", related to different classes of DM-MIMD architectures, were introduced. The libraries can be substituted without modifying the general structure of the compiler.

As regards the implementation templates of the various P^3L constructs, these are particular static networks of communicating processes that, together with the compiling techniques, we consider the *support* of P^3L. In fact, the P^3L programs are compiled in terms of these implementation templates, while the low-level mechanisms of the underlying architecture are employed by the processes making up each implementation template.

The paper is organized as follows. Section 2 briefly introduces the basic features of the P^3L parallel constructs and their composition. Section 3 deals with the characterization of the target parallel machines, and of the corresponding AM. Section 4 discusses the implementation templates of each P^3L constructs and of their composition. Section 5 discusses in detail the overall design of the compiler, and of each of its modules. Section 6 compares our programming methodology to other proposals. Section 7 concludes the paper.

2 The P^3L language

The P^3L language is a high-level, structured, explicitly parallel language. Using P^3L, parallelism can be expressed by means of only a restricted set of parallel constructs, each corresponding to a specific form of parallelism. Sequential parts are expressed by using an existing language, also called the *host sequential language* of P^3L.

From the point of view of programming, the development of parallel application turns out to be easier and more modular. In fact, programmers have to abstract from low-level implementation details, such as process activations, interprocess communications, mapping and scheduling issues. All of these are the responsibility of the compiler, which tunes each implementation taking advantage of the structure given to applications by the use of the P^3L parallel constructs.

However, in so doing, the adoption of a high-level language does not lead to implementations characterized by poor performance. In other words, the current opinion that sees a strict correspondence between high-level programming and poor performance, and between low-level programming and high performance, does not hold in this case.

As regards the current prototype of the P^3L compiler, only a subset of the constructs have been implemented, while the language adopted as host sequential language was C++. The

choice of the constructs to include in the current prototype was made to consider first the most interesting forms of parallelism. C++ was chosen to take advantage of the many tools existing in UNIX for C++, and, above all, because of the success of C++ in the industrial environment for the development of large applications. In fact, existing sequential C++ software can be *reused* within P^3L applications.

The constructs that are currently included in the P^3L prototype compiler, and that can be used for structuring a parallel application, are:

- the `farm` construct, which models *processor farm parallelism*. In this form of parallelism, a set of identical *workers* execute in parallel the tasks which come from an input stream, and produce an output stream of results.
- the `map` construct, which models "easy" *data parallel computations*. In this form of parallelism, each input data item from an input stream is decomposed into a set of partitions, and assigned to identical and parallel *workers*. The workers do not need to exchange data to perform their data-parallel computations. The results produced by the workers are recomposed to make up a new data item of an output stream of results.
- the `pipe` constructs, which models *pipeline parallelism*. In this form of parallelism, a set of *stages* execute serially over a streams of input data, producing an output stream of results.
- the `loop` construct, which models computations where, for each input data item, a *loop body* has to be *iteratively* executed, until a given condition is reached and an output data item is produced.
- the `sequential` construct, which corresponds to a sequential process that, for each data item coming from an input stream, produces a new data item of an output stream.

When describing the various P^3L constructs, we have mentioned some other computations, namely the workers of both the `farm` and the `map`, the stages of the `pipe`, and the body of the `loop`. All of these are, in turn, other P^3L constructs. By means of this mechanism, *hierarchical compositions* of several forms of parallelism can occur. The `sequential` constructs constitute the leaves of the hierarchical composition, because the computations performed by them have to be expressed in terms of the *host sequential language*.

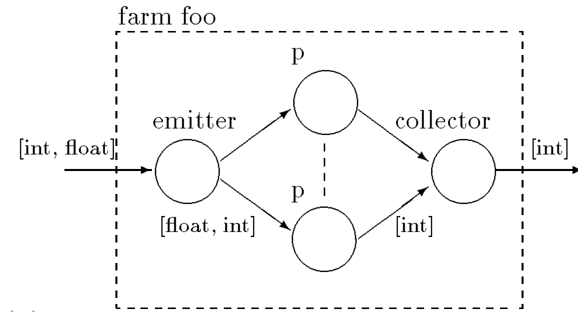
The compositional properties of P^3L rely on the semantics that can be associated with the various P^3L constructs and their hierarchical compositions. In fact, each of them can be thought of as a *data-flow module* that computes (in parallel or sequentially) a function on a given stream of input data, and produces an output stream of results. The lengths of both the streams have to be identical, and the ordering must be preserved, i.e.

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

where $\mathcal{M}$ is the data-flow module corresponding to a generic P^3L construct, $[in_1, \ldots, in_n]$ is the input stream, $[out_1, \ldots, out_n]$ is the output stream, n is the length of both the streams, and every output data item out_i is obtained by applying the function computed by $\mathcal{M}$ on the input data item in_i. The types of the input and the output interface of each P^3L construct, i.e. the types of every in_i and every out_i, have to be declared statically. In fact, the compiler performs the type checking on these interfaces when the P^3L constructs are to be composed.

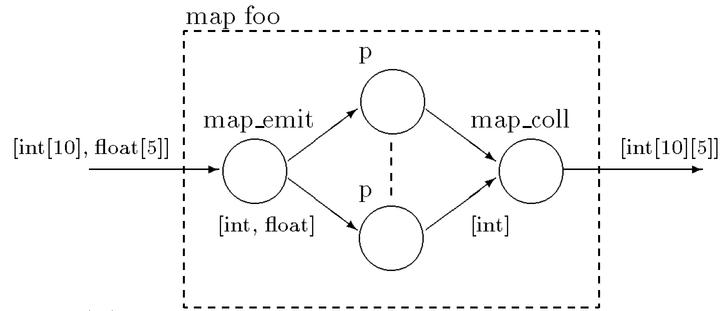
Figure 1 shows the syntax of the P^3L constructs previously illustrated. Next to the various construct syntax, the figure also shows a *network of communicating processes*, called the *logical process structure* of each construct. Even though the logical process structure does not correspond to the actual implementation on the target architecture, it is useful to distinguish the module(s) corresponding to the nested construct(s), as well as the various activities (represented as communication processes) to be supplied by the P^3L support to implement each specific construct.

```
farm foo in(int a, int b)
         out(int c)

   p in(a, b) out(c)

end farm
```



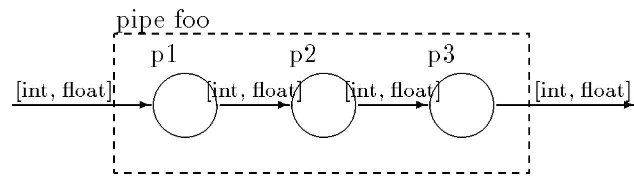
(a)

```
map foo in(int a[10], int b[5])
         out(int c[10][5])

   p in(a[*i], b[*j])
     out(c[*i][*j])

end map
```



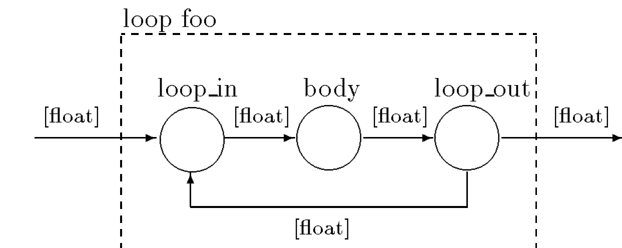
(b)

```
pipe foo in(int a, float b)
         out(int c, float d)

   p1 in(a, b) out(int a1, float b1)
   p2 in(a1, b1) out(int a2, float b2)
   p3 in(a2, b2) out(int c, float d)

end pipe
```



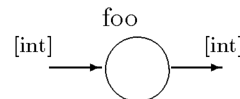
(c)

```
loop foo in(float a) out(float b)
         feedback(b)

   body in(a) out(b)

   until <condition>
end loop
```



(d)

```
foo in(int a) out(int b)
   ${ <seq code> }$
end
```



(e)

Figure 1: Skeletons and logical structures of the P^3L constructs: (a) `farm`, (b) `map`, (c) `pipe`, (d) `loop`, (e) `sequential`.

Figure 1.(a) illustrates the declaration of a `farm` construct. `foo` is the user name given to the `farm` construct, while `p` is the user name of the nested construct. Note that, to declare the `farm foo`, it is necessary to declare the types of the data items composing the input and the output streams, i.e. the *input list* `in(..)` and the *output list* `out(..)` of the parameters. The declaration of the input and the output lists characterizes all the P^3L constructs, as can be seen from Figure 1. The logical structure associated with the `farm foo` shows two processes, i.e. the

emitter and the collector, which perform the distribution of the data and the collection of the results, respectively. They are connected to a set of *workers*, which are instances of the module corresponding to the nested construct p. Programmers do not have to supply the code of any of the distribution and the collection activities, which, in the logical structure, are represented by the processes emitter and collector, respectively. Moreover, also the actual number of workers used in the final implementation must not be specified.

Figure 1.(b) shows a map construct. Each input data is decomposed and passed to each worker p, and the data produced by each worker are recomposed to form a new output data item. The *workers* are instances of the nested construct p. Looking at the logical structure, the process map_emit performs the decomposition and the distribution of the data, while the process map_coll performs the collection and the recomposition of the results. They are connected to a set of workers, which are instances of the nested module corresponding to the construct p. Also in this case, the programmer is not requested to specify the code for the activities represented by the processes map_emit and map_coll, as well as the the actual number of workers to be employed.

Figure 1.(c) shows a pipe construct. It is composed of three stages, corresponding to the constructs p1, p2, and p3. Note the matching between the output type of each stage and the input type of the next one. The logical structure is straightforward.

Figure 1.(d) shows a loop construct. The programmer is requested to specify the call of the nested construct p (i.e., the *loop body*), and the guard that determines when the iterated computation of p has to terminate. The associated logical structure highlights two processes, namely loop_in and loop_out, which perform the iterated call of the nested module p. In fact, p has to take its input data items either from the input stream, or from the stream of the results produced by previous calls of itself. The process loop_out takes the results produced by p, and, in case the final condition has been reached, sends out these results, producing a new item of the output data stream. Whereas, if the final condition has not been reached, since a new call of the nested module p has to occur, the process loop_out sends the results to the other process loop_in, over the feedback channel. A feedback(..) parameter list, appearing in the syntax of the construct, is associated with this channel. Finally, the process loop_in merges the input and the feedback streams. Here too, programmers do not have to specify the code for the processes loop_in and loop_out.

Figure 1.(e) shows a sequential construct, whose user name is foo. As for all the other P^3L constructs, programmers have to declare both the input and the output lists of parameters, while the function computed by the sequential construct must be expressed in terms of the host sequential language. In fact, a piece of sequential code (C++ code, in our case), whose instructions refer to the parameters composing the input and the output lists, has to be supplied. In the syntax, this code appears to be enclosed between two particular brackets, i.e. ${ and }$. The logical structure corresponding to the sequential construct is straightforward, as it consists of a single process with an input and an output channel.

Figure 2 shows the skeleton of a complex P^3L application, in which we can recognize the various P^3L constructs, and their hierarchical composition. The corresponding *construct tree*, which describes the hierarchical composition of the various P^3L constructs, is shown in Figure 3. As an example of composition, note that the construct pipe main (the root of the tree) has three nested constructs (three sons), i.e. farm stage1, loop stage2, and map stage3. The construct tree structure is extensively used by the P^3L compiler for its optimizations.

Figure 4 shows the logical process structure of this program, produced automatically by a tool of the environment from the P^3L source code. The possibility to visualize the logical structure of a complex program, and thus also the composition of various constructs, is very useful during the development of the program.

3 The target architecture and its abstract machine

Since the goal of P^3L is the easy programming of *massively parallel* applications, we have to choose architectures that are able to exploit such kind of parallelism. From the point of view of

```
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 2: The skeleton of a sample P^3L application.

the architectural model, these architectures must be characterized by *scalability* and *bottleneck-freedom*, which require

- homogeneous processing nodes;
- regular interconnection network;
- distributed memory;
- distributed control.

The most suitable architectures are also known as *multicomputers* [18], or DM-MIMD machines. Since the class of the k-ary n-cubes [6] includes many of the most interesting DM-MIMD machines, e.g. two and three-dimensional meshes, hypercubes, etc., this class has been chosen as the target of P^3L. Within our methodology, to guarantee portability of programs, all the members of this class of architectures are viewed through an AM, called P^3M [4]. The interfaces of the mechanisms exported by this AM are uniform for all the members of the class [15].

In our methodology, the mechanisms exported by the AM are not directly used by P^3L programmers, but are exploited by the implementation templates of the various P^3L constructs and their compositions. These templates are used by the compiler to produce the final object

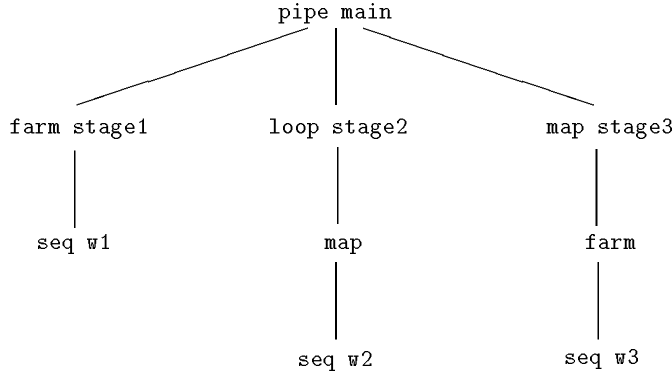


Figure 3: The construct tree of a sample P^3L application.

code. To guarantee the efficiency of porting, the compiler has to choose among these various templates, and has to tune each implementation to better exploit the features of each specific target architecture. The features that the compiler needs to optimize the parallel applications are also exported by the AM, and can be summarized as follows:

- the network topology of the architecture;
- the costs associated with the mechanisms of the AM.

The topology is needed because the compiler has to choose from distinct implementation templates, each supplied for a different interconnection network. The costs are used by the compiler to perform the optimizations of each implementation template, since these costs exactly correspond to the unbound parameters of the performance formulae associated with the template itself.

The AM exports a very reduced set of mechanisms, since only standard sequential operations, process abstraction, and simple message passing between processes allocated to directly connected nodes are provided. We will show how this reduced set of mechanisms suffices to implement the support of P^3L, i.e. the *implementation templates* of the various constructs and their composition. The advantages of using such a reduced set of mechanisms are

- more accurate costs associated with the mechanisms (non-local communications are usually associated with costs that range between a worst and a best case);
- simpler and, at the same time, more effective compiling optimizations. In fact, since the performance models associated with the implementations of each P^3L constructs are very simple, the choices made by the compiler are more effective;

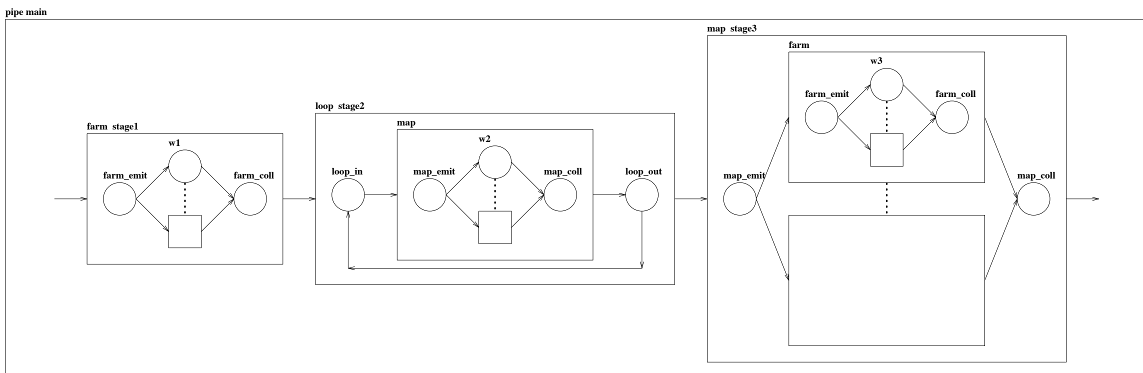


Figure 4: The logical graph of a sample P^3L application.

- simpler and more effective implementations of the mechanisms, as more complex and not frequently used mechanisms are not supplied by the AM;
- better performance of P^3L parallel applications, because they use patterns of parallelism that exploit locality of communications.

The reasons for choosing a reduced set of concurrent mechanisms are similar to the architectural motivations that led to the development of the RISC technology [11].

Of the various k-ary n-cubes, the first architectures we considered as target machines for the P^3L prototype compiler were those which adopt a *two-dimensional mesh* (i.e., the k-ary 2-cube) as a network topology. The mesh is a particular *low-dimensional* k-ary n-cube network, which, because of its simplicity, is easily developed and has been adopted by many of the commercial multicomputers. If different k-ary n-cubes network topologies are compared by taking into account the same technological constraints, the architectures which adopt *low-dimensional* network topologies, e.g. the mesh, are better for exploiting parallel programs characterized by *locality of communications* [7]. Since for each P^3L parallel construct, the compiler accesses implementation templates giving rise to locality-based computations, P^3L applications can take advantage of the mesh-connected DM-MIMD architectures.

4 Implementation templates of the P^3L constructs

When a parallel program has to be allocated to the target architecture, the problem of mapping the process graph onto the processor graph has to be faced. The general mapping problem is known to be $\mathcal{NP}$-hard, and no measure exists to understand how far a solution is from the optimal one. In other words, the problem is also *non approximable* [2,10].

The solution adopted by P^3L consists in restricting the computation model. This restriction led to the definition of a set of parallel constructs whose implementation templates can easily be mapped. These templates can be identified as *locality-based* computations [20], i.e. computations where non-local references are close under some metrics, and are transformed by the compiler into a bounded set of local communications.

The implementation templates discussed below are static networks of communicating processes. The compiler includes the knowledge about these implementation templates, and exploits them to produce the final network of communicating processes, thus implementing the whole P^3L application.

More specifically, the templates presented in this section are related to mesh-based architectures. Distinct implementation templates must be supplied to the compiler if other architectures, adopting different network topologies, are considered as target machines of P^3L.

4.1 Process networks implementing the P^3L constructs

In this section we discuss in detail the implementation templates of the P^3L constructs and their compositions. The templates are those exploited by the current version of the P^3L compiler to generate the final object code for the target machines, and are related to a mesh-based architecture (and, more specifically, to the corresponding AM).

The implementation templates of each P^3L construct will be illustrated by means of directed graphs. Each node of these graphs represents a process that must be allocated, without multiprocessing, to a processing node of the AM, while each arc represents a communication channel that must be assigned, without conflicts, to a bidirectional link of the interconnection network.

The graphs represents the "mapping" of both processes and channels onto a portion of a mesh network (a *sub-mesh*). Below, we will refer to them as the *mapping templates* of the P^3L constructs, as they furnish the "topological information" related to each implementation template. These mapping templates are employed by the compiler to perform those topological optimizations that are needed when several P^3L constructs are hierarchically composed within

a given parallel application. The whole information related to a given implementation template can be obtained by considering together both the mapping template and the actual code of the processes involved (henceforth called *process templates*).

Even though in this section we will present a single implementation template for each construct, the compiler actually includes more than one template for each parallel construct. As we will see in Section 4.2, these templates are different from each other with regard to their topological features, while the communicating processes making up each implementation template are the same.

Furthermore, each implementation template exploited by the compiler to implement a given P^3L construct supplies information about the way in which some specific processes and channels must be arranged and mapped, and, thus, does not correspond to a unique static network of processes. In fact, our methodology relies on the capability of the compiler to modify several parameters of each implementation (parallelism degree, or task granularity), so that, for each specific implementation template, different *implementation instances* can be obtained. Note that, to illustrate the various implementation templates and simplify the presentation, in this section we will use particular instances of each template and of the corresponding mapping.

4.1.1 The `sequential` construct

The actual process that is generated by the compiler for each P^3L `sequential` construct has two channels: an input channel corresponding to the input parameter list, and an output one corresponding to the output parameter list. This process is structured as a sequential loop, in which the process

1. receives a new task to be executed from the input channel;
2. executes the task (the sequential code written by the P^3L programmer);
3. sends the result of the elaboration on the output channel.

4.1.2 The `farm` construct

The logical structure of the `farm` shown in Figure 1.(a) highlights three main activities:

1. the items of the input data stream (on which the tasks have to be computed) arrive at the `emitter` process, and are distributed to the workers;
2. the workers compute the tasks and produce the results;
3. the items of the output data stream (the results of the task computation) are received by the `collector` process, and are produced as output of the whole `farm` construct.

In the actual implementation, these further requirements have been taken into account:

- the distribution activity has to perform the dynamic load-balancing between the workers (due to the variable completion times of the tasks);
- the collection activity has to reorder the task results, in case these results arrive out of order at the `collector`.

The logical structure of Figure 1.(a) cannot be directly mapped onto a mesh. In fact, the degree of each processing node of a mesh is fixed (four communication links), while the fan-out and the fan-in of the `emitter` and the `collector`, respectively, may be greater than four. Thus, the implementation template adopted by the compiler is characterized by a process structure that overcomes this mapping problem, by distributing the activities of the `emitter` and the `collector` to rows of processes. Moreover, the distribution of the `emitter` activity prevents

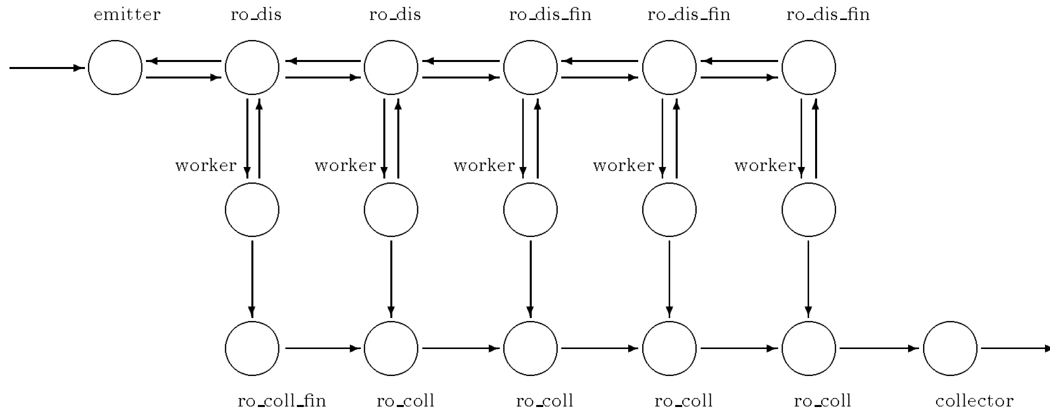


Figure 5: A mapping of a `farm`.

the slow-down due to the, otherwise centralized, scheduling policy. Figure 5 shows a particular instance of the mapping template of the `farm` construct, characterized by five copies of the nested sequential worker.

The `emitter` (emitter process), `ro_dis` (distributing router process) and `ro_dis_fin` (distributing final router process) implement the data distribution for the elaboration of the tasks, and guarantee the load-balancing between the workers. They implement the *task distribution function*.

Each `worker` process receives a data item, computes a task, sends the computed result, and requests a new task. Thus, the `worker` processes implement the *task computation function*.

The processes `collector` (collector process), `ro_coll` (collector router process) and `ro_coll_fin` (collector final router process) implement the collection of the tasks, and guarantee the reordering of the tasks. They realize the *result collection function*.

Implementation of the task distribution The `emitter`, `ro_dis` and `ro_dis_fin` realize the distribution of the tasks by implementing a *slotted ring*. Each slot of this ring may be empty or may contain a task to be computed. The `emitter` process inserts the task into the ring, while `ro_dis` and `ro_dis_fin` extract these tasks, and distribute them to the workers.

No scheduling information is provided for each task, i.e. each slot only contains the data items for the computation of the task, and does not include the name of the worker that will have to execute the task. Moreover, to maintain the input/output ordering, the incoming data items are *tagged* with an increasing mark (*tag*) by the `emitter` process.

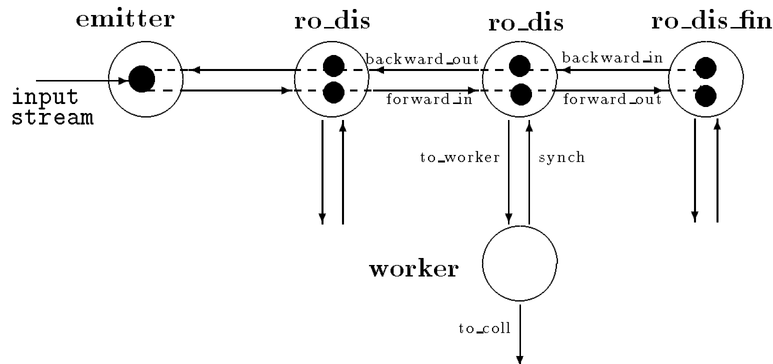


Figure 6: Implementation of the task distribution.

Figure 6 shows the ring in more detail. The black circles in each process represent the slots of the ring. The ring moves along the directions of the channels.

The data items are distributed as follows:

- **Initialization**
  At the beginning, the `emitter`, `ro_dis` and `ro_dis_fin` cooperate for the initial distribution of data. The goal is to fill all the slots of ring.

- **Steady state**
  All the processes that implement the ring communicate with each other, so that the contents of the slots are moved by one step.

  After, all these processes operate on their slots in the following way:

  - if the `emitter` process holds an *empty* slot, it copies an incoming data item (from the input stream) to this slot;
  - if `ro_dis` and `ro_dis_fin` receive (or have received) from the corresponding `worker` a request for a new task, and some of their slots are not empty, they send to that `worker` the contents of a slot.

The slots are emptied as follows:

> if both the slots are full, `ro_dis` and `ro_dis_fin` choose the data item associated with the oldest tag.

The slots are handled in order to maximize the number of full slots present in the ring as follows:

> if the slot corresponding to the backward channels (see Figure 6) is full, and the slot corresponding to the forward channels is empty, the contents of these slots are exchanged. This is done by simply copying some pointers.

This tries to guarantee that most of the slots received by the `emitter` (on the backward channels) are empty, so that, for each movement of the ring, this process is able to inject a new data item into the ring.

Implementation of the task computation The worker of a `farm` (i.e. its nested construct) may be every P^3L constructs. Assume that it is a `sequential` construct. The sequential process implementing the sequential workers of the `farm` is structured as a sequential loop, in which the process

1. receives a new task to be executed from the corresponding `ro_dis` (or `ro_dis_fin`) process;
2. sends a new request for further tasks;
3. executes the task (the sequential code written by the P^3L programmer);
4. sends the result of the computation to the corresponding `ro_coll` (or `ro_coll_fin`) process.

Note the difference between the implementation of a `sequential` construct when it is nested in a `farm`, and the implementation discussed above, to be used when, for instance, the `sequential` construct is a stage of a `pipe`. In fact, when this construct is nested in a `farm`, the corresponding process needs, besides the input and the output channels, a new channel, i.e. the `synch` one shown in Figure 6. This new channel is used to request a new task to the processes which implement the distribution of the tasks.

The same channel is also necessary when the nested construct of the `farm` is a parallel one. In this case, the nested construct will be implemented by a process network, and the process that receives the input stream for this network will need a `synch` channel to request a new task.

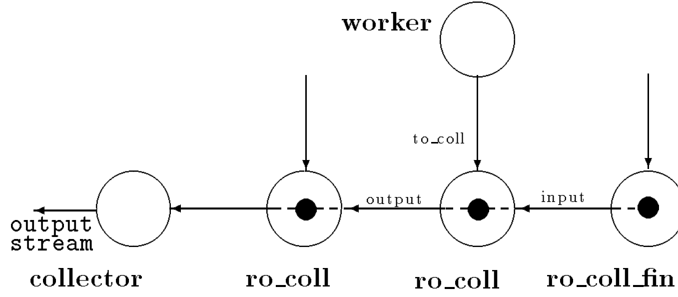


Figure 7: Implementation of the collections of the task results.

Implementation of the collections of the task results The `collector`, `ro_coll` and `ro_coll_fin` collect the task results. They implement a *slotted chain*, where each slot may be empty or may contain a result to deliver to the `collector` process.

Figure 7 shows these processes and the channels between them. The black circles represent the slots. The chain moves following the direction of the channels. If a `ro_coll` receives a result from the corresponding worker, and its slot is empty, this result is copied into the slot. The slot of the `ro_coll_fin` is always empty, because it cannot be filled by previous processes in the chain.

In order to maintain the output stream ordered, the following policy is adopted

> if a `ro_coll` process receives a result, its slot is occupied by another one, and the tag of the former is smaller than the tag of the latter, the two buffers are swapped. The results with smaller tags thus advance first.

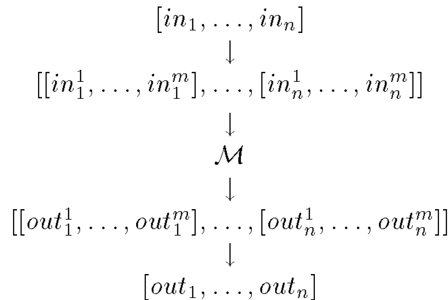
The `collector` receives the contents of a slot at each movement of the chain. If the received slot is not empty, its contents are inserted into an ordered structure. This avoids sending out the results of the tasks with a different order from the input stream order.

Moreover, while a slot is received, the `collector` may also send out a previously received result, which is stored in the ordered data structure. This only occurs if all the task results associated with smaller tags have been sent beforehand.

4.1.3 The `map` construct

The implementation template for the `map` construct is derived from that of the `farm`. In fact, we can see the `map` as a module that, for each input data item, produces sub-streams (*data decomposition*) of independent tasks, which are concurrently executed, while the results corresponding to these sub-stream are combined to form a single output data item (*data recomposition*).

The following diagram shows how the `map` implementation works, where $\mathcal{M}$ represents the module corresponding to the nested construct of the `map` (i.e. the *worker* of the `map`).



where n is the length of both the input and the output streams, while m is the number of partitions into which each input and output data item has to be decomposed. In fact, in_i and out_i, $i \in \{1, \ldots, n\}$, represent generic data items of the input and the output data streams, respectively, while in_i^j and out_i^j, $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, represent data partitions of these data items.

An instance of the mapping template of the `map` construct is thus the same as that shown in Figure 5. The only difference is that the `emitter` and the `collector` processes are replaced by `map_emit` and `map_coll`, respectively. In fact, besides the task distribution, and the ordered collection of the results, `map_emit` has to implement the *decomposition* of the input data, i.e. the transformation $in_i \rightarrow [in_i^1, \ldots, in_i^m]$ for all $i \in \{1, \ldots, n\}$, while `map_coll` has to implement the *recomposition* of the output, i.e. the transformation $[out_i^1, \ldots, out_i^m] \rightarrow out_i$ for all $i \in \{1, \ldots, n\}$.

The data packets that flow from `map_emit` to the workers, and from the workers to `map_coll`, include other information concerning the specific partitions of the input and the output data, respectively. In other words, a tag i is associated with each partition in_i^k and out_i^k to distinguish the task, and another tag k is associated with them to distinguish the specific partition.

4.1.4 The `pipe` construct

To generate the mapping template of a `pipe` construct, the processes implementing the various stages of the `pipe` are glued together by means of channels. Note that, if the stages are not sequential processes, the implementations corresponding to them are, in turn, other networks of processes.

Figure 8 shows the mapping of a `pipe` construct composed of nine sequential stages. Note how processes and channels are placed so that the area of the mesh occupied by the final mapping template is characterized by

- regularity, since a two-dimensional squared box encloses this area;
- no waste of computing resources, since all the processing nodes of the box are allocated for executing some process.

A heuristic is used by the compiler to allocate the processes that belong to distinct stages and need to communicate - i.e. the processes that produces the output stream for each stage, and the processes that receives the input stream for the next stages - as close as possible (e.g. on neighboring processing nodes). Sometimes, special routing processes are added. These processes simply by-pass the data received from the input channel to the output one.

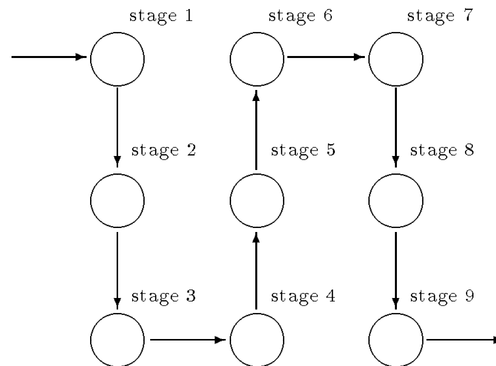


Figure 8: A mapping of a `pipe` construct composed of nine sequential stages.

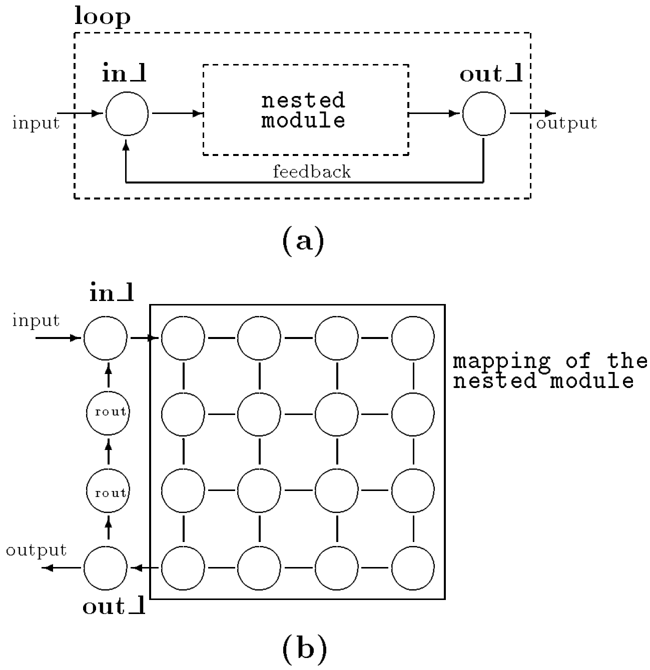


Figure 9: (a) Logical structure of a `loop` construct - (b) A mapping of the `loop` and of the nested construct

4.1.5 The `loop` construct

The goal of the implementation template of the `loop` construct is to allow the tasks coming from both the `input` and the `feedback` channels of the loop to be concurrently computed. Since distinct tasks can be concurrently in execution, tasks are tagged (distinct tasks have distinct associated *colors*).

To illustrate how this implementation works, let us consider Figure 9, which shows the logical structure and the mapping of a `loop` construct.

The `in_l` process implements the merging between the tasks flowing over the `input` and the `feedback` channel, which is implemented as a chain of router processes. `in_l` performs a nondeterministic choice between the two channels, but, to avoid deadlock, the priority of the guard associated with the `feedback` channel is greater than the priority of the other guard.

The `out_l` process receives the results from the nested module, and checks whether the final condition has been reached. It sends the result of the task either

> over the `feedback` channel, if the final condition is not reached,

or

> over the `output` channel of the `loop` if the final condition is reached. The ordering of the stream is maintained by using an ordered structure, in the same way as in the `collector` process of the `farm` does. The tagging used to distinguish (to color) the tasks is also used to keep the output stream ordered.

4.1.6 Termination

A special function is performed by the process that reads from the input data stream of the whole P^3L application. When this process encounters the end of the input stream, it sends a special last packet, called the *end-of-stream* packet.

The processes implementing the `sequential` constructs of the P^3L application check whether the received input data item is the *end-of-stream* packet. If it is, the packet is by-passed, and is not computed.

The *end-of-stream* packet is used to start the distributed termination of the parallel program. This function is performed by the process that produces the output data stream of the whole P^3L application. It begins the distributed termination of the program on the reception of the *end-of-stream* packet, but only after the reception of all the packets with smaller tag than the *end-of-stream* one.

4.2 Composition and mapping

In Section 2, we discussed the problem of composing P^3L constructs at the language level. In this section, we discuss the same problem with respect to the mapping templates adopted by the compiler for each P^3L construct.

As regards the mappings presented in the previous section, they are related to an AM with mesh topology. To cope with the composition issues, the compiler associates with each of them a 2-dimensional box, with a pair of input/output channels. This box is the smallest one that encloses the process graph implementing the construct.

If the compiler has to deal with architectures characterized by different interconnection network topologies, the dimension number of the boxes associated with the various mapping templates is different. For example, if the architecture belongs to the class of the k-ary 3-cubes, i.e. the interconnection network is a *three-dimensional mesh*, the various mapping templates are associated with *three-dimensional boxes.*

The association of a box with the process network implementing each P^3L construct is used by the compiler to maintain, in a parametric way, the knowledge on each mapping template. In fact, the parameters of each mapping template are exactly the features of boxes corresponding to the nested constructs. For example, Figure 5 shows a mapping of a `farm` with five workers, where each worker is a sequential process. This is the limiting case of the general *mapping template* of the `farm`, shown in Figure 10, where the workers are regarded as (identical) boxes with the input and the output channels connected to processes placed on two opposite sides of the box. Note the new processes, `ro_dis_mid` and `ro_coll_mid`, which bypass the data flowing on the distributing ring and the collecting chain, respectively.

For this first prototype of the compiler, we have restricted the shapes of the boxes allowed. Figure 11 shows the possible shapes of the boxes, which we call *box templates.* They have a pair of input/output channels placed either on two opposite corners, or on the two corners of the same side of the box. Moreover, they may have different height/width ratios. In order to compose the P^3L constructs, the compiler may also consider rotating and flipping each box template (i.e. the processes and the channels contained in the box).

The next section shows how several mapping templates are supplied to the compiler for each P^3L construct. These mapping templates are different in the characteristics of the enclosing box templates. The compiler chooses from different mapping templates for the same construct by considering the features of the box templates.

As an example of a different mapping template for a `farm`, if we change the orientation of the collection chain in the mapping template in Figure 10, we obtain another general mapping template, where the input and the output channels are both placed on the same side of the enclosing box. Note that, to realize this new mapping template, no new processes need to be introduced besides those presented above.

4.3 The analytical performance models

Until now we have discussed the mapping templates of the various P^3L constructs. The information about these templates is accessed by the compiler, which chooses the most suitable instance of the selected mapping template for a given P^3L construct. This choice is made by tuning the

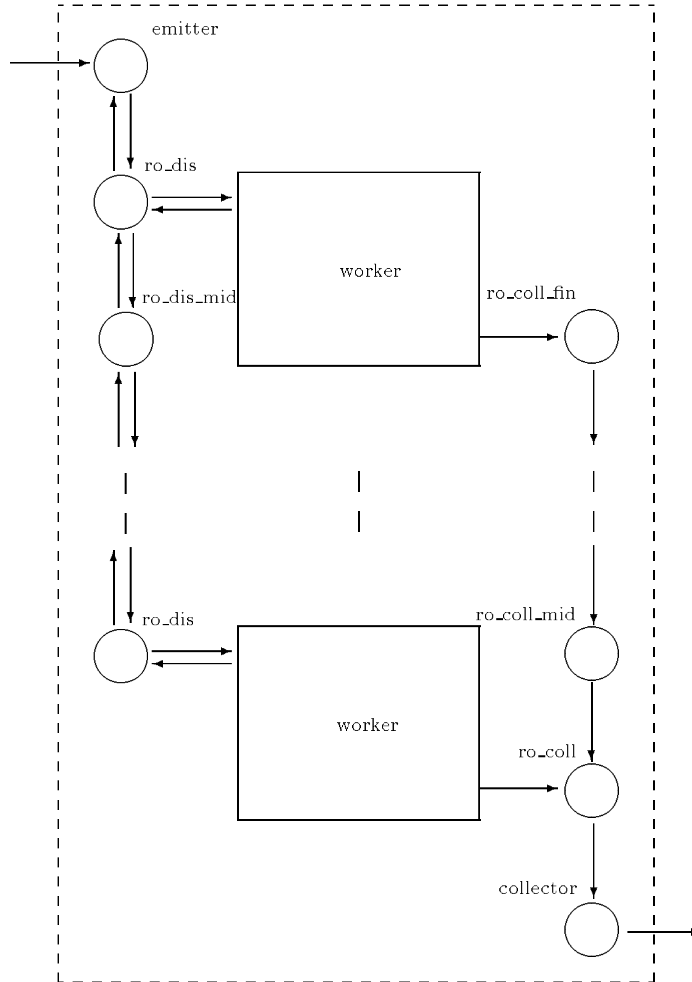


Figure 10: A *mapping template* of the `farm`.

expected performance of the final implementation, deciding both the degree and granularity of the parallelism exploited.

For example, once the compiler has selected the mapping template in Figure 10 for a `farm` construct, it also has to choose the particular instance of the mapping template, i.e. it has to decide the actual number of *workers* to include in the final implementation. The compiler takes this kind of decision by accessing the costs associated with the mechanisms of the AM, and considering, at the same time, the performance associated with the mapping(s) of the nested construct(s).

The goal of the performance tuning task performed by the compiler consists in minimizing the *completion time* (or, equivalently, maximizing the *speedup*) of each implementation in computing a set of tasks [17]. The tasks are related to the data items composing the input stream of the data-flow modules corresponding to the various P^3L constructs and their composition. This kind of optimization also corresponds to maximizing the *bandwidth*[1] of each implementation, provided that the input data stream is long enough to make the parallelism exploited useful. In fact, all the implementations associated with the various P^3L constructs are characterized by a transitory time interval, at the end of which the maximal bandwidth is achieved.

In this section we introduce the *analytical performance models* which are associated with each mapping template, and which allow synthetic formulae to be derived and used by the compiler.

[1] The bandwidth corresponds to the number of tasks that are completed within one unit of time.

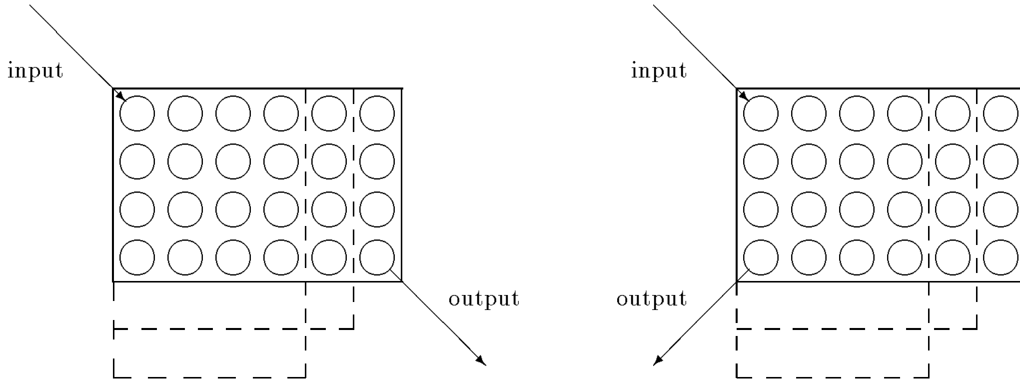


Figure 11: Possible *boxes templates* that can enclose the various mapping templates.

These formulae are parametric with respect to the costs exported by the AM, as well as the bandwidth of the nested module(s) of the construct.

As an example, we outline the analytical model used to derive the performance formulae associated with the `farm` construct. These formulae are used

- to devise the useful parallelism degree (i.e. the number of workers in the `farm`);
- to return the bandwidth of the final implementation of the `farm`.

As discussed in Section 4.1.2, in the implementation of the `farm` we can distinguish three pipelined stages:

1. the *distribution* stage, composed of an emitter process along with the processes implementing the slotted ring;
2. the *computation* stage, composed of a set of identical workers;
3. the *collection* stage, composed of an collector process along with the processes implementing the slotted chain.

In order to maximize the bandwidth, these stages have to run in parallel and have to be balanced. These requirements are exactly those needed for optimizing the bandwidth of a general pipeline structure (in fact, the same model is used to attain the performance and to optimize the implementation of the `pipe` construct).

To define the formulae that model the `farm` implementation, the following costs must be considered:

- τ_p, which is the average time spent inserting an input data item into the distributing ring (or extracting an output data item from the collecting chain) and moving the distributing ring (or the collecting chain) by one step.

  This time depends on many costs exported by the AM mechanisms. For instance, τ_p corresponds to the time spent to copy a data item in an empty slot, and to perform a message exchange between neighboring processing nodes. These times depends on the costs of copy and communication mechanisms (AM-dependent), expressed as a function of the data item size (problem-dependent);
- β_w is the average bandwidth of each worker. If the worker is a sequential process, and τ_w is the time taken by the worker to compute a single task, we have that $\beta_w = \frac{1}{\tau_w}$.

By using these costs, we can derive that

- the bandwidth of the distributing (collecting) stage is $\frac{1}{\tau_p}$, because one data item is inserted into (extracted from) the distributing ring (collecting chain) at each time interval τ_p;
- the bandwidth of n_w workers (n_w turns out to be the parallelism degree of the implementation) is $n_w \beta_w$.

Since we want to balance the bandwidth of the three functional stages of this `farm` implementation, the following equation must hold:

$$\frac{1}{\tau_p} = n_w \beta_w$$

From this equation we can derive the former formula, stating the best parallelism degree that can be exploited usefully by this `farm` implementation, i.e.

$$n_w = \lceil \frac{1}{\tau_p \beta_w} \rceil \tag{1}$$

In case the workers are sequential processes, the formula (1) becomes $n_w = \lceil \frac{\tau_w}{\tau_p} \rceil$.

Roughly speaking, the number of workers n_w determined by (1) may be thought of as the *useful parallelism degree* that this implementation is able to exploit. In fact, if we used more workers, the distributing ring would not be able to keep all the workers busy, while, if we employed less workers, sometimes some tasks would not inserted in the distributing ring because the slots are still full.

The latter formula associated with this `farm` implementation corresponds to the rate by which the input (output) stream is computed (produced), and thus corresponds to the total bandwidth of the `farm`:

$$\beta_{\texttt{farm}} = \frac{1}{\tau_p} \tag{2}$$

In order to apply the formula (1), the compiler needs to know the bandwidth of the workers, namely β_w. In particular, if these workers are sequential processes, the time τ_w taken by the workers to compute a single task has to be known. In any case, only an average measure of τ_w needs to be known. This average is used to find the optimal number of workers n_w. The implementation of the distribution stage by means of a slotted ring performs the dynamic scheduling of the tasks and guarantees the load balancing even if there is a very large variance of τ_w.

We have done tests on a `farm` with sequential workers, where τ_w was uniformly distributed on a given interval. The number of workers n_w was found by using (1) w.r.t. to the average value of τ_w, i.e. the midpoint of the distribution interval. The tests showed that, also in this case, the worker load are perfectly balanced.

Figure 12 shows the profiling execution of the three workers of a `farm` construct. This diagram is one of the graphical statistical outputs that can be visualized by the emulator of parallel architectures [16], on top of which our AM, namely P^3M, has been implemented. Note that the tasks of the `farm`, represented by the *user code* bars, do not take the same time to execute. In fact, the execution times range *uniformly* over a given time interval.

The result on the best parallelism degree for the implementation of each construct is very strong. In fact, if the input stream can only be read sequentially by a single process (the emitter), we state that there exists an upper bound on the achievable *speedup*, which is our best number of workers n_w. The speedup of this implementation does not scale with the dimension of the problem, i.e. with the length of the input stream. We have found that, even when the input stream becomes longer, since it is not useful to add further workers, then the actual speedup does not change.

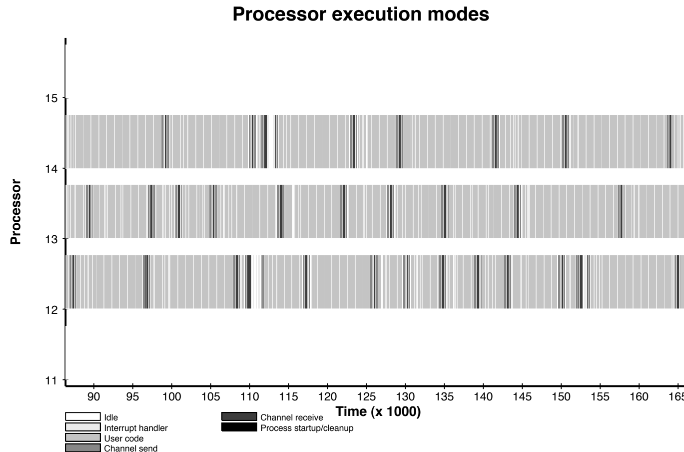


Figure 12: Example of load balancing obtained by a `farm` implementation.

Only under some hypotheses, can we augment the size of the packets distributed to the workers (in this case, a packet contains several tasks) in order to add more workers and attain better speedups [17]. This is a form of *grain* optimization, which we are going to exploit in the next version of the P^3L compiler. Anyway, also when adopting grain optimizations, there exists an upper bound on the number of workers that can be exploited usefully, i.e. on the best achievable speedup.

Similar formulae are used by the compiler for the other P^3L constructs. The `map` formulae are the same as those used for the `farm`, but take into account the further overheads derived from the decomposition/recomposition tasks. The `pipe` formulae are exploited to balance the granularity of the various stage, while the final bandwidth is the smallest one of the bandwidths of all the stages. The `loop` formulae take into account the recursive call of the nested module, and the overheads incurred in the non-deterministic merging performed by the `in_l` process, and in the final condition check performed by the `out_l` process.

5 The structure of the compiler

In the previous section, we discussed in detail the implementation templates of the various P^3L constructs. The compiler maintains the information concerning these implementation templates and the strategies to access them within a set of *libraries*. This library-based organization is useful for reducing the parts of the compiler that need to be changed when other architectures, with different interconnection network topologies, are considered as possible targets of P^3L.

The most important library accessed by the compiler is called the *mapping library*. It concerns the topological aspects of the implementation templates of each P^3L construct (also called *mapping templates*), as well as the *analytic formulae* associated with them.

The *mapping library* is accessed by the compiler through a set of *rules*. The goal of these rules is to optimize the implementation by driving the compiler during the selection of the best mapping template for each construct. Other rules are used to transform the construct tree associated with each specific application, in case the compiler realizes that either some parallel

construct introduces useless parallelism, or further parallelism can be exploited by inserting other parallel constructs. All these rules are arranged in another library, called the *optimization library*.

The last library accessed by the compiler is called *process templates library*. It contains the sequential processes (i.e. the actual code for these processes, written by using the host sequential language and the AM mechanisms for concurrency) which implement the mapping templates included in the *mapping library*. For example, the `emitter` and the `collector` are two of the templates included in the *process templates library*.

To discuss in more detail how all these libraries are actually exploited by the compiler, it is useful to refer to the general design of the compiler, and particularly, to the *front-end*, the *middle-end*, and the *back-end*, which make up the P^3L compiler:

1. The *front-end* parses the source code, checks the types, and produces an internal data structure, representing the *construct tree* of the application. An example of this tree is shown in Figure 3. Each node of the tree is annotated with information about the parameter passing between the hierarchically composed constructs, and, in case of the `sequential` constructs, with the names of the files and of the corresponding procedures containing the user-provided code (which, in our case, is written in C++).

2. The *middle-end* processes the information contained in the construct tree, and produces an internal data structure, mainly stating

   - the names of processes and channels,
   - the mapping of processes and channels onto an abstract representation of the target architecture,
   - the types of the data flowing over the channels.

   The *middle-end* accesses the mapping templates of each P^3L construct. These templates are provided parametrically by the *mapping library*. The mapping templates refer, in turn, to some sequential processes, whose actual code in supplied by the *process templates library*.

   Each entry of the *mapping library* is associated with some performance formulae. These formulae are parameterized with respect to the costs exported by the AM, and are used to optimize each implementation (in terms of the parallelism degree and the granularity of the parallel activities) and to return its bandwidth (i.e. the number of tasks produced in a given time interval).

   Access to the *mapping library* is guided by a set of rules, contained in the *optimization library*. These rules call the mapping entries taking into account the composition patterns found in the construct tree. Some other rules are used to transform the construct tree in case the parallelism expressed by some construct is not useful for improving the speedup, or further parallelism can be exploited by introducing other parallel constructs.

3. The *back-end*, taking the mapping data structures produced by the middle-end, generates the actual code for the AM. This task is carried out by defining and mapping both processes and channels, using the mechanism interface provided by the AM.

   In particular, predefined processes, included in the *process templates library*, are used to generate the code of each process. The types of data to be transmitted over the channels, and some special functions (e.g., for copying between the specific data structures associated with the channels) have to be directly supplied by the back-end for each of these processes.

The general design of the P^3L compiler is shown in Figure 13, which also illustrates the interactions of each part of the compiler with the libraries and the AM.

5.1 The libraries

We chose a library approach to the design of the compiler to simplify and make more modular and modifiable the general structure of the compiler, and, above all, the structure of the *middle-end*

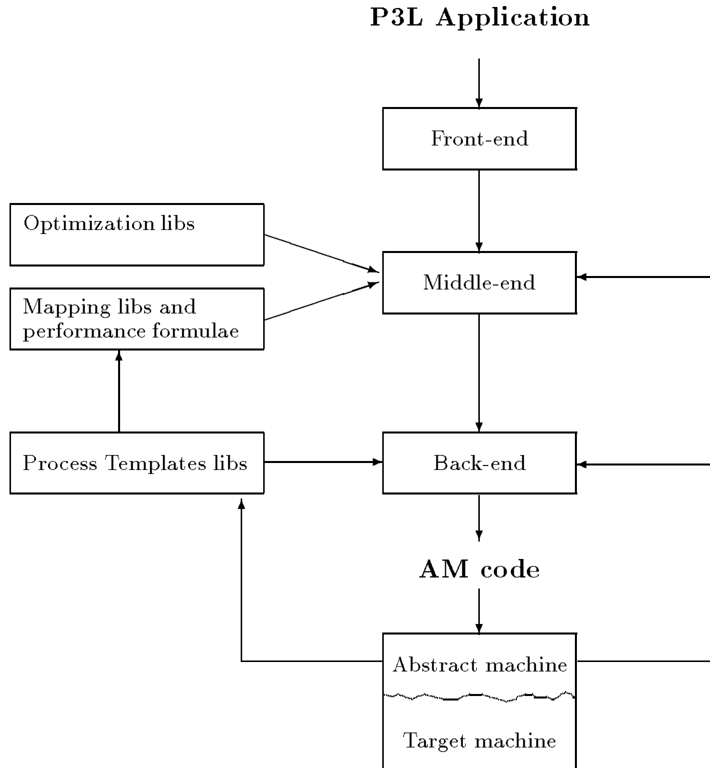


Figure 13: The structure of the P^3L compiler.

part, which contains all the innovative optimization techniques that our programming methodology has made possible. In addition, the library organization of the template processes make the *back-end* part independent of possible new constructs added to the language.

Below we describe in more detail the libraries, assuming that they all relate to an AM exporting a given network topology. As previously discussed, for the current prototype compiler, we only developed the libraries for the two-dimensional mesh topology. Only supplying other libraries, related to AMs exporting different network topologies, can the compiler produce object code for other kinds of architectures.

Mapping library The *mapping library* contains many entries for each P^3L construct, each corresponding to a different mapping template. Each entry of the library can be selected by means of the following tuple:

$$< \mathcal{C}, \mathcal{N}, \mathcal{B}, \mathcal{A} >$$

where

- $\mathcal{C}$ is name of the construct (e.g. `farm`, `pipe`, etc);
- $\mathcal{N}$ is the data structure representing the mapping of the nested construct (in case $\mathcal{C}$=`pipe`, since the `pipe` has several nested constructs, corresponding to the various stages, more of these mapping structures are included in $\mathcal{N}$). As discussed above, at this level the nested constructs are thought of as box templates with some constraints on the input/output channels;
- $\mathcal{B}$ corresponds to the bandwidths of the nested construct (if $\mathcal{C}$=`pipe`, $\mathcal{B}$ includes the bandwidths of several nested constructs);

- $\mathcal{A}$ is a set of attributes on the shape of the requested mapping, i.e. the shape of the box template that will have to enclose the process structure implementing the construct $\mathcal{C}$;

The call of each library entry returns a pair $(\mathcal{M}, \mathcal{B}')$, where

- $\mathcal{M}$ is the data structure representing the final mapping of the construct $\mathcal{C}$. This mapping will be enclosed in a given box template. Note that the same data structure $\mathcal{M}$ can be used (as an $\mathcal{N}$ parameter) to query the library if the construct $\mathcal{C}$ is, in turn, nested in another P^3L construct;
- $\mathcal{B}'$ gives the bandwidth of the final mapping. Note that it can be used (as a $\mathcal{B}$ parameter) to query the library if the construct $\mathcal{C}$ is, in turn, nested in another P^3L construct.

The various entries of the *mapping library* access to different mapping templates, and to the performance formulae associated with them. Using these formulae, along with the bandwidths $\mathcal{B}$ of the nested construct, and the costs exported by the AM, each library entry is able to produce the optimized implementation for the construct $\mathcal{C}$.

Both the data structures $\mathcal{N}$ and $\mathcal{M}$ refer to a set of predefined process templates (e.g., `collector`, `emitter`, `ro_dis`, `loop_in`, `loop_out`, etc.). These processes are, in turn, included in another library, the *process template library*.

Process template library The code corresponding to each process template is written by using the *host sequential language*, plus the concurrency mechanisms exported by the AM. Basically, this code refers to

- a set of channels, which are used by the communication mechanisms;
- a set of data structures, which corresponds to the data types of the channels.

Depending on the specific process template, different operations may be performed on these data structures.

For example, `map_emit` has to decompose the input data structure, which is transformed into a collection of other (smaller) data structures. Unfortunately, all these operations differ because of the type associated with the data structures in distinct `map` constructs. So, to make the code of the template independent of the specific construct, macros are used for each operation. In order to complete the code of each process template, the *back-end* of the compiler has to supply the macro definitions. Since the same kind of operation recurs in several process templates, the compiler only has to be able to provide a few types of macro definitions.

Optimization library This library, used by the compiler to optimize the implementation of the P^3L applications, contains a set of *rules*. These rules are exploited for selecting the most suitable entries from the *mapping library*, as well as for transforming the construct tree when some of the constructs introduce parallelism which is of no use with respect to that specific target architecture (e.g., the grain of parallelism is too fine for that architecture). Distinct *optimization libraries* must be provided for architectures based upon different network topologies.

The two kinds of rules included in the *optimization library* can be formally described by

$$\frac{\mathcal{P} \qquad \mathcal{M}}{\mathcal{O}pt_{Map}} \tag{3}$$

$$\frac{\mathcal{P} \qquad \mathcal{M}}{\mathcal{O}pt_{Tree}} \tag{4}$$

where

- $\mathcal{P}$ is a precondition concerning the bandwidth of the nested construct(s). This precondition has to hold in order to apply the rule;
- $\mathcal{M}$ is a precondition concerning the structure of the construct tree. In particular, $\mathcal{M}$ gives takes into account the parent of the construct to be mapped. Thus, $\mathcal{M}$ may be regarded as a sort of pattern matching over the tree, which has to hold in order to apply the rule;
- $\mathcal{O}pt_{Map}$ corresponds to an action that is activated only if both $\mathcal{P}$ and $\mathcal{M}$ hold. $\mathcal{O}pt_{Map}$ annotates the construct tree in such a way that, later on, a specific entry of the *mapping library* can be selected. It also determines how this entry has to be called, i.e. the attributes $\mathcal{A}$ used to query the library.
- $\mathcal{O}pt_{Tree}$ also corresponds to an action that is activated only if both $\mathcal{P}$ and $\mathcal{M}$ hold. It is concerned with the transformation of the piece of the construct tree identified by the matching rule $\mathcal{M}$.

5.2 The front-end

The *front-end* of the compiler parses the P^3L part of the source program, ensures that it is syntactically correct, and performs the type checking on the input/output parameters lists of each P^3L construct involved.

In addition, the pieces of C++ code provided by the programmer for each `sequential` construct are included in C++ procedures, stored in special files. These files are separately compiled using the host language compiler to check their syntactical correctness.

The *front-end* was realized by using the standard `lex` and `yacc` tools. Its output is the construct tree of the program, which describes the hierarchical composition of the P^3L constructs. This tree is exported as a set of Prolog data structures. In fact, for easy prototyping, the other parts of the compiler have been written in Prolog.

Figure 14 shows the output of the *front-end* for a simple P^3L program. For each construct, we have a Prolog fact, called `construct()`, that specifies the name, the kind of the P^3L construct, and the types of the input and output parameters. The `called()` component included in this fact is used to define the hierarchical structure of the construct tree.

Moreover, for each construct there exists a Prolog fact that defines either

- the parameter passing to the nested constructs (e.g., the Prolog fact `farm(..)` in Figure 14), or
- the host language files including the user-provided code. This only occurs for the `sequential` constructs (e.g., the the Prolog fact `seq(..)` in Figure 14).

The most interesting Prolog facts are the `seq(..)` ones, which are associated with the `sequential` constructs. These facts are structures composed of several fields. The first field is the user-name of the `sequential` construct. The second is the call of a C++ procedure including the code that the user has specified in `${ }$`. The third field identifies the file that includes the C++ procedure. The fourth field corresponds to other host language modules to be linked together in order to obtain the final process (in this example, we have the `m.C` module, which includes the definition of the function `f()` used by the user-provided code). Finally, the fifth field defines particular host language libraries to be used to produce the final process (in this case, no libraries have been specified by the user).

5.3 The middle-end

The *middle-end* takes the construct tree generated by the *front-end*, and produces some other Prolog structures, determining which processes and channels are employed to implement a given P^3L application, as well as mapping them onto the target architecture.

```
w in(int a) out(int b)
    ${
       int f(int);
       b = f(a);
    }$
    src(m.C)
end


farm f in(int x) out(int y)
    w in(x) out(y)
end farm
```

$\overset{front-end}{\Longrightarrow}$

```
construct('w',
  type(seq),
  inlist([var_formal('a', int)]),
  outlist([var_formal('b', int)]),
  called([])
  ).

construct('f',
  type(pure_farm),
  inlist([var_formal('x', int)]),
  outlist([var_formal('y', int)]),
  called(['w'])
  ).

seq('w', 'p3l__0__', './p3l__0__.C',
  src(['m.C']),
  libs([])).

farm('f', [
  call('w',
    inlist([var('x', actual)]),
    outlist([var('y', actual)]))
  ]).
```

Figure 14: Example of parsing a simple P^3L program.

The algorithm adopted by the *middle-end* is based on a *depth-first visit* of the construct tree associated with each P^3L application. In fact, this algorithm visits the tree from the leaves to the root, and, for each node of the tree, corresponding to a given P^3L construct, selects and calls an entry of the *mapping library*, using the rules included in the *optimization library*. In some cases, instead of selecting a *mapping library* entry, the rule employed modifies the construct tree.

Now we show how the *middle-end* algorithm works by means of an example. Suppose that, during the visit of the construct tree, the *middle-end* algorithm has to map a `farm` construct. Since the various constructs are mapped from the leaves to the root of the tree, when the *middle-end* algorithm encounters the `farm` construct, the descendants of the `farm` have already been mapped. Therefore, we can suppose that the workers of the `farm` have already been mapped, and have been associated with a given *box template*, whose input and output channels are placed on two opposite corners.

To better understand the different strategies used by the *middle-end* algorithm, it is now useful to consider two different construct trees. In the former tree, our `farm` is nested in a `loop` construct, while, in the latter, it is nested in a `pipe` construct. Figure 15 shows the portions of the two different trees, showing the `farm`, the parent constructs, and the nested constructs (i.e. the workers), here represented by the associated *box template*.

Several entries of the *mapping library* may be called to select a given mapping template for the two `farm` constructs. These entries differ with respect to the kind and the shape of the box template that will enclose the final mapping. The *middle-end* algorithm uses the rules of the type (3), included in the *optimization library*, to select the most suitable entry.

If the parent of the `farm` is a `loop`, as in the example illustrated in Figure 15.(a), the selected rule produces a final mapping whose box template has the input and output channels placed on the same side of the box. In fact, looking at the mapping template of the `loop` construct (shown in Figure 9), we can see how the goal of this rule is to attain a `feedback` channel (implemented by a chain of routing processes, and going from the `loop_out` to the `loop_in` process) as short as possible.

Whereas in the case illustrated in Figure 15.(b), where the parent of the `farm` is a `pipe`, the selected rule produces a final mapping whose box template has the input and output channels placed on two opposite corners. This kind of mapping simplifies the heuristics strategy for the successive mapping of the `pipe`.

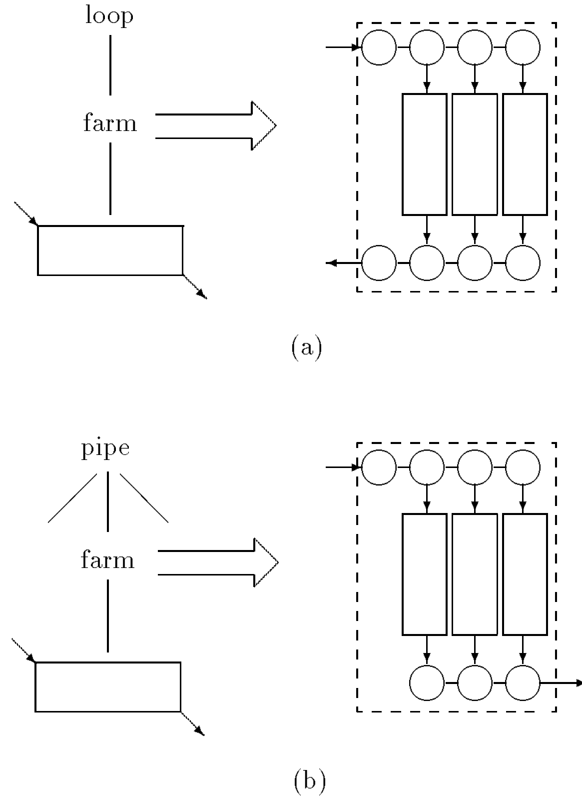


Figure 15: A mapping of a `farm` depending on the type of its father, (a) `loop`, or (b) `pipe`.

In the two examples above we applied two distinct rules of the type (3), which differ in the $\mathcal{M}$ preconditions, and, of course, in the $\mathcal{O}pt_{Map}$ actions. In the former case, we applied a rule where the matching precondition $\mathcal{M}$ is `loop(farm)`, i.e. the rule can be applied if the `farm` is the nested construct of a `loop`. In the latter case, the matching precondition $\mathcal{M}$ is `pipe(farm)`, i.e. the rule can be applied if the `farm` is the nested construct of a `pipe`. On the other hand, the $\mathcal{P}$ preconditions are the same in both the rules. They are concerned with the bandwidth of the implementation of the nested construct, i.e. the worker of our `farm`, and only hold if its bandwidth is less than $\frac{1}{\tau_p}$. Note that $\frac{1}{\tau_p}$ is the best bandwidth that could be obtained by our `farm` implementation. In other words, the $\mathcal{P}$ preconditions only hold if the bandwidth of the worker is *low enough* to make it useful to exploit `farm` parallelism.

Thus, if no $\mathcal{P}$ preconditions hold for all the rules of the type (3), the *middle-end* algorithm searches for an applicable rule from those of the type (4), characterized by the same $\mathcal{M}$ preconditions (they relate to the same sub-tree), but whose $\mathcal{P}$ preconditions are the negation of those appearing in the rules of the type (3). Thus, due to this properties, a rule of the type (4) will be selected only if the bandwidth of the worker of the `farm` is very high (i.e., more than, or equal to $\frac{1}{\tau_p}$). Since, in this case, the `farm` parallelism is not useful, the $\mathcal{O}pt_{Tree}$ action corresponding to the selected rule modifies the construct tree of the P^3L application, removing the `farm` construct from the tree. Figure 16 shows the tree-to-tree transformations for both the examples in Figure 15. This kind of transformations preserves the semantics of the P^3L programs.

Mapping data-structures produced by the middle-end During the visit of the tree, each time the *middle-end* performs the mapping of a construct, it produces some Prolog data structures, henceforth called *mapping data-structures*.

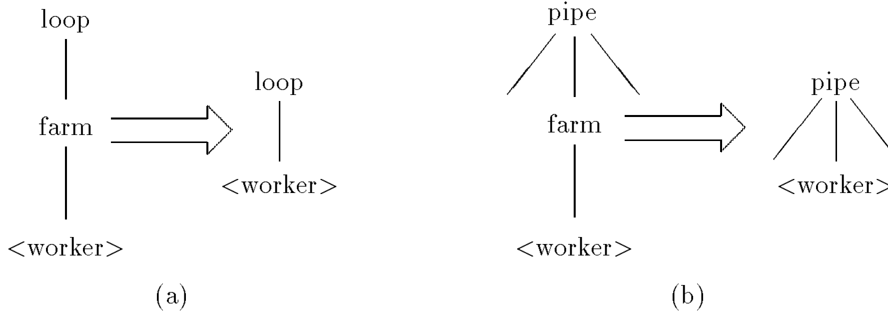


Figure 16: Tree transformations involving a `farm` construct, whose father is (a) a `loop`, or (b) a `pipe`.

A very small subset of these structures states the mapping of processes and channels onto nodes and links of the mesh-based target architectures, while the other structures are bound with this subset by means of special identifiers.

Flippings and rotations that may be needed to map the various constructs only change the subset of the data-structures which state the mapping of process and channels. Furthermore, if more instances of the same construct are to be inserted into a mapping (e.g., because that construct is, in turn, a worker of a `farm`), only this subset of data structures need to be replicated.

Consider the simple program shown in Figure 14. The *middle-end* receives the tree representation of the program, and produces the mapping shown in Figure 17, deciding, at the same time, the best parallelism degree (in this case, the number of workers has been decided to be three). This figure was automatically produced by a tool of the environment, using the *mapping data-structures* produced by the *middle-end*.

The following is a small subset of all the data structures produced, and describes the mapping:

```
map(f, farm,
    param([proc_in(0, 0)],
          [proc_out(2, 0)],
          [link_io(mitt(0, -1), dest(2, -1))],
          ....
         ),

    [ map_real(proc(0, 0), proces(emitt354),
         param([ring_pos(3)]),
         [receive(proc(0, -1), in, type349),
          receive(proc(0, 1), back_in, type352)],
         [send(proc(0, 1), forw_out, type352)]),

      map_real(proc(2, 0), proces(coll355),
         param([]),
         [receive(proc(2, 1), coll_bus_in, type353)],
         [send(proc(2, -1), out, type350)]),

      map_real(proc(0, 1), proces(ro_dis367),
         param([ring_pos(2)]),
         [receive(proc(0, 0), forw_in, type352),
          receive(proc(0, 2), back_in, type352),
          receive(proc(1, 1), synch, type351)],
         [send(proc(0, 2), forw_out, type352),
          send(proc(0, 0), back_out, type352),
          send(proc(1, 1), in, type346)]),
```

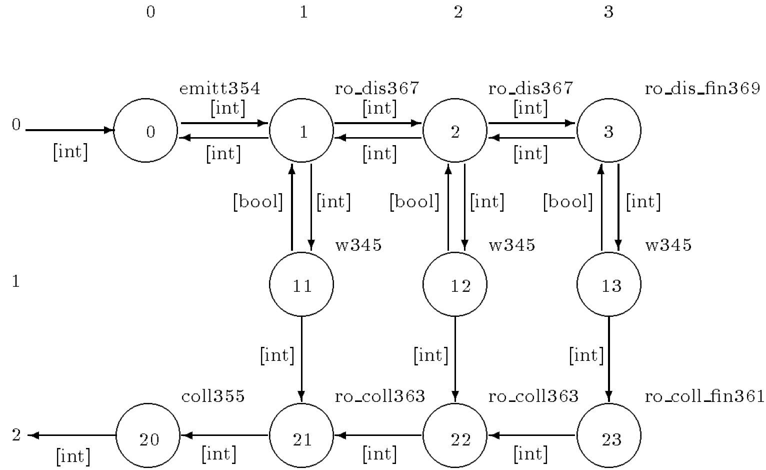


Figure 17: A mapping of a P^3L program.

```
    ....
  ]
).
```

It consists of a `map(..)` Prolog fact, which contains a list whose elements are, in turn, `map_real(..)` structures describing the mapping a single process along with its channels.

Let us describe the various components of a `map_real(..)` structure:

```
map_real(proc(Ind1, Ind2),
         process(Templ_id),
         param([..]),
         [receive(proc(Ind1_in, Ind2_in), Channel_name_rec, Type_id), ...],
         [send(proc(Ind1_out, Ind2_out), Channel_name_send, Type_id), ...])
```

The first component, `proc(Ind1, Ind2)`, identifies the processing node onto which `process(Templ_id)` (appearing as the second component of the structure) is mapped. `Ind1` and `Ind1` are two numerical indexes identifying a given processing node of a 2-dimensional mesh[2]. The `Templ_id` key is used to identify all the processes that are identical as far as code and channel types are concerned.

The third component of a `map_real(...)` structure determines specific parameters to be passed to each process. In the example above, a constant parameter is to be passed to each process that implements the distributing ring of the `farm`. This parameter states the relative position of each process in the ring, and is exploited by these processes for the initialization.

The fourth and the fifth components are two lists, describing the channels used to receive and send messages, respectively. Note that, since each channel connects the node `proc(Ind1, Ind2)` with another neighboring node, a `proc(Ind1_in, Ind2_in)` (`proc(Ind1_out, Ind2_out)`) information is associated with each *receive* channel (*send* channel). In addition, symbolic names are associated with the channels, i.e. `Channel_name_rec` and `Channel_name_send`. These names are the ones used in the actual code of the processes to refer to these channels. Finally, an identifier `Type_id` is associated with each channel. `Type_id` identifies the type of the data to be sent over the channel.

Besides the mapping data structure presented above, the *middle-end* produces other information, bound with this structure by means of the identifiers `Templ_id`, and `Type_id`. In fact, the

[2]The number of indexes needed to identify a given processing node depends on the number of dimensions of the specific k-ary n-cube network topology.

types of the data to be transmitted over the channels are specified by further Prolog facts with the following structure:

```
type_def(Templ_id, Type_id,
    type(back_end_inf(...),
         type_task(...)) ).
```

For each `Type_id` related to a given `Templ_id`, there are two kinds of data types:

- Problem dependent data, defined by the `type_task(..)` component. These data correspond to the input and the output lists of each P^3L construct, as specified by the programmer.
- Implementation dependent data, defined by the `back_end_inf(..)` component. Some examples of implementation dependent data are the `tags`, associated with each task to maintain the ordering of the input/output stream, or the `end_mark`, used to distinguish the *end-of-stream* packet.

The following are a subset of the `type_def(..)` structures produced for our example:

```
type_def(emit354, type349,
      type(back_end_inf([], end_mark),
           type_task([int]))).
type_def(emit354, type352,
      type(back_end_inf(fe_mark, [tag], end_mark),
           type_task([int]))).
type_def(ro_dis367, type352,
      type(back_end_inf(fe_mark, [tag], end_mark),
           type_task([int]))).
....
```

Moreover, since each `Templ_id` must be associated with a process included in the *process template library* (see Figure 13), the *middle-end* also produces other Prolog facts such as the following:

```
library_code(Templ_id, Code_id).
```

This fact relates a `Templ_id` with a `Code_id` key, which can be used, in turn, to access the *process template library*.

The following are a subset of the `library_code(..)` structures produced for our example:

```
library_code(emit354, emitter).
library_code(coll355, collector).
library_code(ro_dis367, ro_dis).
library_code(ro_dis_fin369, ro_dis_fin).
....
```

Note that, in the previous Prolog structures, we used the `Templ_id` keys to distinguish the various processes, instead of directly using the `Code_id` keys. Suppose that a P^3L application includes more `farm` constructs. This implies that we also have more `emitter` processes, whose channels are, in principle, of different types. If we had used the same `Code_id` key to identify these processes, we could not have associated distinct `type_def(..)` facts with each of the `emitter` processes.

The following are some of the entries of the *process template library* that are selectable by means of the `Code_id`'s:

```
template_seq(emitter, farm_emitter,
             'farm_emitter.C', src([]), libs([]),
             macros( ... )).

template_seq(collector, farm_collector,
             'farm_collector.C', src([]), libs([]),
             macros( ... )).

template_seq(ro_dis, farm_ro_dis,
             'farm_ro_dis.C', src([]), libs([]),
             macros( ... )).

template_seq(ro_dis_fin, farm_ro_dis_fin,
             'farm_ro_dis_fin.C', src([]), libs([]),
             macros( ... )).
.....
```

Each entry of this library identifies the process call, the file that contains the code of the process, and other source modules and host language libraries that have to be linked together. Moreover, a set of macro definitions which must be supplied by the *back-end* are identified. For each of these macros, which are not presented here, some other Prolog facts are provided by the *middle-end*.

5.4 The back-end

The last part of the compiler is the *back-end*, which heavily depends on the interface provided by the AM. The library-based structure given to the compiler has allowed us to make this part independent of the new constructs added to the language. In fact, the *back-end* takes the *mapping data structures* produced by the *middle-end*, and performs a per-process translation, taking into account the *process template library* entries involved.

The translation process performed by the `back-end` produces the static configuration of the target AM, as well as the actual code for the process (with the relative channels) to be mapped on each processing node. The AM for which the *back-end* has been developed, namely the P^3M AM [15], has a configuration language that resembles the Prolog *mapping data structures* produced by the *middle-end*.

The translation process performed by the *back-end* consists of the following parts:

- determine the *static mapping* of processes and channels. For each processing node, the *back-end* determines the identifier of the process to be mapped on, while, for each physical link, it determines the channels and the corresponding data types to be allocated. Since our AM is usually programmed by a compiler, the types associated with the channels are raw blocks of data of a given size. The type checking on the data to transmitted over the channels is assumed to be performed by the compiler, before producing the code for the AM;

- determine the specific channel and constants to be passed as parameters to the processes to be mapped onto the various nodes. Notice that, since more instances of the same process are usually mapped on different nodes, these channels and constants make different processes that are instances of the same template;

- produce the actual code of each process, determining the host-language modules and libraries to be linked together. For this task, the *back-end* exploits the *process template library*, whose entries correspond to the `template_seq(..)` Prolog facts. The *back-end*'s task consists of completing the code of each process, adding the definitions of the data structures and the macros used inside this code.

  Note that the way in which the code for every process is produced does not depend on the kind of P^3L construct that the process implements. Thus, this makes the *back-end* part

independent of the new constructs added to the language. In fact, when a new construct is added to P^3L, the *mapping library* has to be enriched with other mapping templates, referring, in turn, to new processes to be included in the *process template library*.

6 Related work

The research track illustrated in this paper is similar to that followed in the field of the *universal models for parallel computation* [22,20]. In that case, however, the goal was to find a model that can be "efficiently ported" to different kinds of architectures, where the efficiency of the porting is exactly measured in terms of the cost of each implementation, expressed as product of the execution time of the program and the number of processing units employed.

Our approach also tackles the problem of efficiency of porting, but to a greater extent. In fact, our programming methodology guarantees that each program implementation exploits the greatest degree of parallelism that the specific target architecture is able to exploit, without trying to achieve the universality on those architectures that have been chosen as possible targets.

Moreover, since our programming methodology considers the problem of easy parallel programming, our research is similar to a novel approach to the exploitation of parallel machines by using functional languages [9,20]. In that case, in fact, the idea consists of encapsulating certain common algorithmic forms in higher-order functions to facilitate parallel programming development. This approach, being based on clear semantic properties, also allows program transformation technologies to be devised, and makes it possible to derive a program implementation from an initial specification using algebraic identities between higher-order functions. These algorithmic forms are also called *skeletons*, as defined in [5], which proposes some implementation of these skeletons on a two-dimensional mesh architecture.

The approach followed by [5], however, does not consider the issue of composing basic forms of parallelism, as its skeletons reflect some specific strategies employed to implement particular *combinations* of more basic forms of parallel computation. On the other hand, the methodology proposed by [9] is more general, as each skeleton has a definite declarative semantics. In fact, the skeletons are defined as polymorphic, higher-order functions, and this makes the application specifications independent of any particular implementation. However, the process of transformation of a high-level specification of an application into a set of lower-level skeletons relative to a particular target machine (or, using our terminology, into a combination of implementation templates) is mainly the programmer's responsibility, though some performance models and rules of transformation help this process.

The higher-order functions considered in [9] correspond to both data and control parallel computations, while those considered in [20] are relative to only the data parallel model, which restricts the control flow to only a single thread expressed as sequence or function composition [19].

If we consider other programming methodologies proposed to make the process of the parallel software development easier, we have to consider the coordination languages and the corresponding computational models. *Linda* [1], which is the most important example of this kind of language, provides the abstraction of a shared, content-addressable memory that can be accessed by any process. While Linda is architecture independent, and thus holds those characteristics of high-levelness that facilitate the parallel programming job and the portability of programs, the tuning of each Linda application for each specific target machine is the responsibility of the programmer. Often, the code must be restructured to better exploit the specific features of each machine.

This lack of efficiency in porting previously coded applications is also the principal drawback of the high-level programming environments based on graphical development tools. Usually these kinds of environments provide an interface for systems which, by means of a set of libraries and run-time software, allows a heterogeneous system to be seen as a unified, virtual, parallel machine. The most important example of these types of systems is PVM [21]. The same considerations made for Linda also hold for PVM. Although PVM may allow transparent, architecture

independent programs to be developed, performance tuning is always the responsibility of the programmer, and often program restructuring is needed to better exploit the specific features of a new target system.

7 Conclusions and future work

This paper has discussed a new approach to parallel programming of general purpose, massively parallel, architectures.

The main innovative features of our programming methodology is the introduction of primitive language constructs, which allows programmers to easily express different patterns for the exploitation of parallelism. Since these constructs can be composed, a complex application can be "structured" by using several of these constructs. Furthermore, different techniques of parallelization can easily be tested, by simply structuring the parallel application by means of different constructs.

Another important feature is the easy programming of parallel applications, since the information that programmers are requested to supply concerns the "quality" of the parallelism to be exploited, while all the implementation details are the responsibility of the compiler and its support. This approach guarantees the portability of programs, provided that the compiler is able to produce code for the new target architecture. Furthermore, the possibility of recognizing the parallel structure of the application has allowed new compiling techniques to be devised. These techniques, which perform the tuning of several implementation parameters, guarantee the efficient exploitation of the features of each target architecture.

The object code produced for the various parallel constructs and their compositions exploits locality of references. Locality is one of the main requirements to achieve high performance, and allows architectures to be expanded without consequences to the scalability of the program implementations. Other approaches to parallelism do not exploit locality at all, and sometimes non-local references depend on the size of the architecture (e.g. the latency of non-local communications, which, in turn, depend on the actual parallelism degree of the machine).

The prototype of the P^3L compiler was developed during a joint project, called P4 [3], involving the Department of Computer Science of the University of Pisa, and the Hewlett Packard Laboratories-Pisa Science Center.

Within this project, the P^3M AM interface was implemented on top of an emulator of parallel architectures [16,14], which is able to produce performance profiling figures for each program run. This emulator can be configured, so that different costs can be associated with each mechanism of the AM, and different interconnection network topologies can be tested. The emulator was very effective in validating the performance models associated with the various P^3L constructs and their composition, and thus, to show the feasibility of the general methodology adopted in the project.

We are currently devising implementation templates, performance models, and process templates for the other parallel P^3L constructs that are currently lacking. These other constructs model

- `tree` computations, and, more specifically, a `map` followed by a `reduce` operation. The `reduce` consists of the application of an associative operation on a vector $\mathcal{T}[\,]$ ($\mathcal{T}$ is a P^3L type) to produce a single item, with $\mathcal{T}$ type;
- `geometric` computations, i.e. *data-parallel* computations where data are decomposed in a geometric way to vectors or arrays of processing nodes, and some limited exchanges of data between "neighbors" occur during computation;
- `MISD farm` computations, where distinct functions are computed for each input data item, i.e. distinct P^3L constructs are to be provided as nested modules.

Further future research will consist of porting the environment on an actual parallel DM-MIMD architecture, namely the Meiko-Computing Surface. This should involve porting our AM interface on the target machine.

Other activities on P^3L are concerned with porting the language and its environment to architectures with different network topologies from the *mesh*. This work requires theoretical studies on the implementation templates of the various P^3L constructs, and their compositions. Taking into account the same technological constraints, we hope that it will be possible to compare the architectural models with respect to their ability to run parallel application structured as hierarchical compositions of P^3L constructs. One possible outcome might be to identify some classes of network topologies and corresponding architectures on which the forms of parallelism identified by the P^3L constructs can be implemented more efficiently.

Acknowledgments

We would like to thank Milon Mackey, for implementing the front-end of the P^3L compiler, and for the useful discussion about the interface with the host sequential language (C++) of P^3L.

We also thank Mark Syrett, who implemented some parts of the middle-end of the compiler, and the graphical tools of the environment.

Our research also benefited from discussions with Roberto Di Meglio, who was one the first users of the P^3L programming environment.

Finally, we would like to thank the Hewlett Packard Laboratories, which supported this research, and all the P4 team at the HP Pisa Science Center.

References

[1] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnswamy. Matching Languages and Hardware for Parallel Computation in the Linda Machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[2] S. Antonelli and S. Pelagatti. On the Complexity of the Mapping Problem for Massively Parallel Architectures. *Int. Journal of Foundation of Computer Scinece*, 3(3):379–387, 1993.

[3] 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 *Proc. of COMPSAC 91*, pp. 536–543, Tokyo, Japan, 1991.

[4] F. Baiardi and M. Jazayeri. An Abstract Machine for Highly Parallel Architectures. Technical Report HPL-PSC-92-37, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.

[5] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitmann/MIT Press, 1989.

[6] W. Dally. Performance Analisys of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39:775–785, June 1990.

[7] W. Dally. Network and Processors Architecture for Message-Driven Computers. In R. Suaya and G. Bithwistle, editors, *VLSI and Parallel Computation*, Chap. 3, pp. 140–222. Morgan Kaufmann Publisher, Inc. - San Mateo, California, 1991.

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

[9] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, R.L. While, and Q. Wu. Parallel Programming Using Skeleton Functions. Technical report, Dept. of Computing, Imperial College of Science, Technology and Medicine, London, May 1992. Draft.

[10] D. Fernandez-Baca. Allocating Modules to Processors in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-15(11):1427–1436, November 1989.

[11] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann Publisher, San Mateo, California, 1990.

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

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

[14] M. Mackey. The `p3m` command. Technical Report HPL-PSC-92-48, Hewlett Packard Laboratories, Pisa Science Center (Italy), September 1992.

[15] M. Mackey and T. Sullivan. P^3M machine interface definition. Technical Report HPL-PSC-92-43, Hewlett Packard Laboratories, Pisa Science Center (Italy), August 1992.

[16] M. Mackey and T. Sullivan. Proteus user manual (PA-RISC). Technical Report HPL-PSC-92-44, Hewlett Packard Laboratories, Pisa Science Center (Italy), August 1992.

[17] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa - Dipartimento di Informatica - Italy, March 1993. TD-11/93.

[18] C.L. Seitz. Concurrent Architectures. In R. Suaya and G. Bithwistle, editors, *VLSI and Parallel Computation*, Chap. 1, pp. 1–83. Morgan Kaufmann Publisher, Inc. - San Mateo, California, 1991.

[19] D. B. Skillicorn. Models for Practical Parallel Computation. *Int. Journal of Parallel Programming*, 20(2):133–158, April 1991.

[20] D.B. Skillicorn. Architecture-Independent Parallel Computation. *IEEE Computer*, pp. 38–50, December 1990.

[21] V.S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[22] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.