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

Abstract

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 be efficiently 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, using 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 interface is used by the compiler to produce the final object code.

Keywords. Parallel Processing, Parallel Languages, Compilers, DM-MIMD Architectures.

1 Introduction

Although a lot of parallel architectures exist on the market, a programming methodology and/or a computational model that allow programs to be efficiently ported to different machines is still a subject of research. In particular, a big challenge is the definition of a programming methodology for massively parallel processing that is characterized by ease of use, and guarantees portability without needing heavy interventions of programmers to tune the performance of each application.

Whilst ease of use depends on the high-level features of a parallel language, 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, and 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*) [10],
- a set of compiling tools that, exploiting the structured features of P^3L, realizes the efficient portability of applications, and
- an abstract machine (AM), called P^3M[5], 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 ignore low-level implementation details, such as process activations, interprocess communications, mapping and scheduling issues, 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 are the *pipeline*, the *processor farm*, the *map*, etc. P^3L supplies distinct language constructs for each of them. These constructs can be hierarchically composed to express, in a *structured* way, more complex forms of parallel computation.

The P^3L constructs are the *only* means that a programmer has to give *parallel structure* to an application. We have been supported in our choice by the evolution of sequential programming languages. In fact, a look at the history of imperative languages shows the progressive discarding of elementary constructs, e.g. `goto`'s, in favour of a set of constructs with definite control flow behaviour, e.g. `while`, `if`, `for`, etc. The same evolution can be observed in field of the data-structures. The introduction of these new constructs/data-structures has not only improved programmers' 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 low-level parallel constructs from P^3L, such as `send/receive` or `parallel process activation`, in favour of a set of high-level 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*.

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 and related works [14, 17]. 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. These forms of parallelism and their compositions can be efficiently implemented on different massively parallel architectures. These implementations, henceforth called *implementation templates*, embody many strategies to *statically* solve typical issues like mapping, scheduling, degree and granularity of parallelism, and also to solve problems deriving from hierarchical composition of different P^3L constructs. Furthermore, they can be modelled analytically, so that their performance can be estimated at compile time.

In our programming methodology, the efficient portability of programs relies on the compiler, which adopts a specific *template-based* approach allowed by 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, and uses these templates to generate the final code.

An implementation template corresponds to a parametric network of processes, along with mapping information on an abstract network topology, and analytic performance formulae. The low-level mechanisms of the underlying architecture are employed by the processes making up each implementation template. The formulae associated with the templates are parameterized with respect to the costs of the same mechanisms. For this purpose, in our methodology, each target machine is viewed through an AM, which exports the interface of the mechanisms to the compiler along with the associated costs. The performance formulae are used to optimize the final implementation, and, thus, to choose the best instance of the possible process networks proposed by the templates.

The aim of this paper is to discuss in more detail the overall design of the P^3L compiler, and, more specifically, we want to describe the implementation of our prototype compiler. The prototype works for a subset of all the P^3L constructs, and produces code for a mesh-based AM that exports a very reduced set of mechanisms. The final code runs on an emulator of parallel architectures [20, 18], and on a real parallel machine, a Meiko CS1 based on T800 Transputers.

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 presents the implementation templates of the P^3L constructs and of their composition. Section 5 describes the overall design of the compiler, and of each of its modules. Section 6 compares our programming methodology to other proposals.

2 The P^3L language

The P^3L language is a high-level, structured, explicitly parallel language. Using P^3L, parallelism can be expressed only by means of 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.

The P^3L constructs can be *hierarchically composed* to express more complex forms of parallelism. This compositional property relies on the semantics associated with the various P^3L constructs and their 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. In the following we often use the term *input task* in place of input data item, to emphasize the fact that a new instance of a given task is "fired" by the incoming data item.

Another feature of P^3L is its interface with the host sequential language. The interface has been designed to make easier portability between different host languages. In fact, sequential parts are completely encapsulated into the constructs of P^3L. Parameter passing between P^3L constructs are handled by linguistic constructs that are external to the specific host sequential language, while the data types that can be used to define the interface of the P^3L constructs are a subset of those usually available in the most common and widespread languages.

In the current prototype of the P^3L compiler, the language adopted as host sequential language is C++. C++ has been 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 are listed below along with their informal semantics. Other proposed constructs are presented in [10], while their formal semantics is discussed in [11].

- The `farm` construct, which models *processor farm parallelism*. In this form of parallelism, a set of identical *workers* execute in parallel the independent 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` construct, which models *pipeline parallelism*. In this form of parallelism, a set of *stages* execute serially over a stream 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.

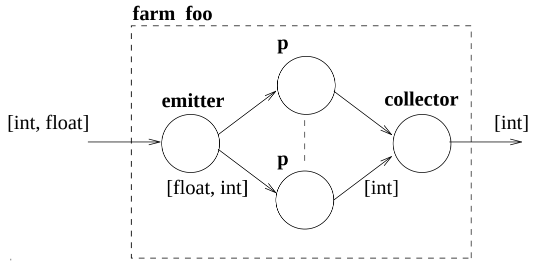
When describing the various P^3L constructs, we have mentioned some other computations, namely the workers of both the `farm` and the `map`, the stages of the `pipe`, and the body of the `loop`. All of these are, in turn, other P^3L constructs. Thus hierarchical compositions of several forms of parallelism 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*.

Figure 2 shows the syntax of the P^3L constructs described above. Next to the various constructs' 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 particular communicating processes) to be supplied by the P^3L support to implement each specific construct.

Figure 2.(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. As we can see from Figure 2, the declaration of the input/output lists must be provided for all the P^3L constructs. 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. Moreover, the actual number of workers used in the final implementation must not be specified, since it will be devised by the support taking into account the characteristics of the target architecture and of the user code.

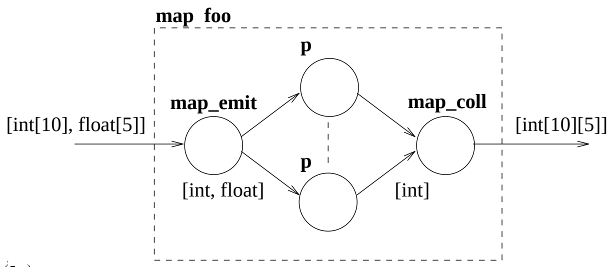
Figure 2.(b) shows a `map` construct. Each input data is decomposed and passed to each worker `p`, and the data produced by each worker are recomposed to form a new output data item. The *workers* are instances of the nested construct `p`. Looking at the logical structure, the process `map_emit` performs 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 requested to specify neither the code for the activities represented by the processes `map_emit` and `map_coll`, nor the actual number of workers to be employed.

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

   p in(a, b) out(c)

end farm
```



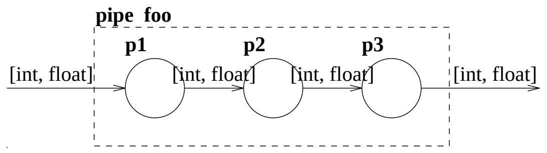
(a)

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

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

end map
```



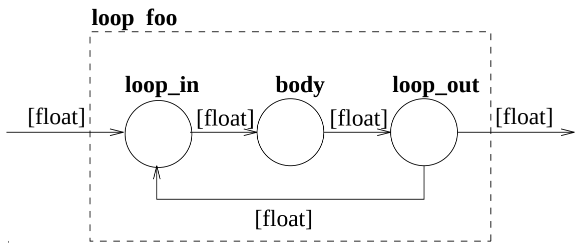
(b)

```
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(c, d)

end pipe
```



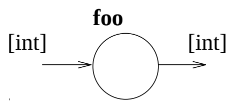
(c)

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

Figure 2.(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 2.(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 specifying when the iterated computation of `p` terminates. The associated logical structure shows 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 (thus 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 activities represented by `loop_in` and `loop_out`.

Figure 2.(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 fragment 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 P^3L brackets, i.e. `${` and `}$`. This code can access the data structures in the input/output lists as if these structures have been declared in the same block as the fragment of code. Alternatively, the sequential code could be specified by naming a function and the module where this function has been defined. Moreover, even if this is not the case of the example of Figure 2.(e), it is possible to define, inside the body of the `sequential` construct, other modules and libraries to be linked together to obtain the final process (an example of this is shown in Figure 10). The logical structure corresponding to the `sequential` construct is straightforward, as it consists of a single process with an input and an output channel.

An example Now we present an example, discuss the computational steps that may be exploited to solve it, and show how these steps can be naturally translated into a P^3L program. Suppose that we have a stream of tuples, each composed of $N+1$ floating point numbers: $\{y_0, \ldots, y_N\}$. Each tuple corresponds to the y coordinates of $N+1$ points belonging to the graph of a continuous function. The x coordinates of these points, i.e. $\{x_0, \ldots, x_N\}$, are considered to be fixed, as the functions have been sampled at fixed intervals. For each tuple $\{y_0, \ldots, y_N\}$, we want to plot a differently coloured graph corresponding to a continuous function passing through all the points identified by the tuple. To plot the intermediate points whose x coordinates are contained in each sample interval $[x_i, x_{i+1}]$, we determine first a polynomial function $f : ax^3 + bx^2 + cx + d$. The coefficients $\{a, b, c, d\}$ are determined by imposing the passage of f through the points (x_i, y_i) and (x_{i+1}, y_{i+1}), and, moreover, by equating the two derivatives $f'(x_i)$ and $f'(x_{i+1})$ with a pair of constants. The constant corresponding to each point x_i is the average of the slopes of a pair of linear functions, the former passing through (x_{i-1}, y_{i-1}) and (x_i, y_i), and the latter passing through (x_i, y_i) and (x_{i+1}, y_{i+1}). For the two end-points (x_0, y_0) and (x_N, y_N), we use the slopes of the two linear functions passing thorough each point and, respectively, the next and the previous one. Therefore, the computational steps to display a curve can be summarized as follows:

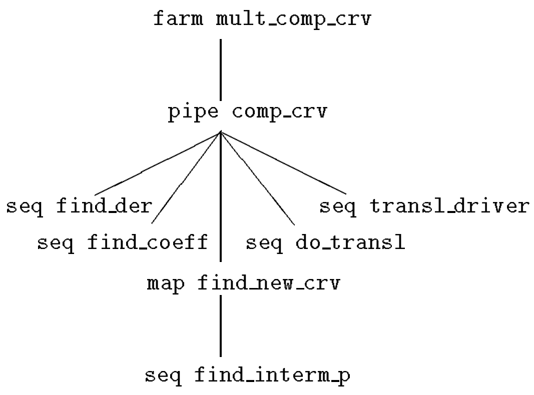
1. Find the derivatives in correspondence of the points $\{(x_0, y_0), \ldots, (x_n, y_n)\}$.

2. Determine the coefficients a, b, c, d of the polynomial functions corresponding to each of the n intervals.

3. By using the polynomial functions found in the previous step, determine the intermediate points needed to draw a "continuous" curve on the screen.

```
pipe comp_crv in(float y_tpl[N+1], int colour)
              out(char crv[OBJ_SZ])
  find_der in(y_tpl, color)
           out(y_tpl, colour, float der[N])

  find_coeff in(y_tpl, colour, der)
             out(colour, float a[N], float b[N],
                        float c[N], float d[N])

  find_new_crv in(colour, a, b, c, d)
               out(colour, float new_y_tpl[N][100])

  do_transl in(colour, new_y_tpl)
            out(colour, float adj_y_tpl[100*N])

  transl_driver in(colour, adj_y_tpl)
                out(crv)
end pipe


farm mult_comp_crv in(float y_tpl[N+1], int colour)
                   out(char crv[OBJ_SZ])
  comp_crv in(y_tpl, color) out(crv)
end farm
```

(a)



(b)

Figure 2: (a) A sample P^3L application - (b) its construct tree.

4. Perform the translation of the axes, and convert the coordinates of the points with respect the new unit of measure. In fact, the Cartesian system to which our curve refers must be represented on the specific Cartesian system associated with the screen.

5. Translate the curves in a representation that can be processed by the driver of the screen.

The five steps illustrated above can be easily written as a P^3L `pipe`. Moreover, since the computation of each curve is independent of each other, a `farm` structure can be specified as the outermost form of parallelism, where the `pipe` is employed as a *worker* of the `farm`. Figure 2.(a) illustrates this application. Note that some parameters, e.g. `colour` in the first four steps, are simply by-passed to the following steps without being modified. For the sake of brevity, we only show the P^3L construct calls, and omit the declaration of the constructs that are used as steps of `pipe comp_crv`. It is worth noting that these steps can be either sequential constructs, or, in turn, other parallel constructs. In general, however, a P^3L programmer should specify as much parallelism as possible. In fact the compiler does not parallelize sequential code, but decides about which and how much of the parallelism specified by the programmer must be actually exploited to obtain effective final implementations. For example, the step `find_new_crv`, which, for all the intervals $\{[x_i, x_{i+1}] \mid i = 0, \ldots, N-1\}$, computes the ordinates of `M` points, could be expressed by means of a `map` construct, whose nested construct is a `sequential` one, i.e. `find_interm_p`, which computes only the points of a single interval:

```
find_interm_p in(float c1, float c2, float c3, float c4)
              out(float new_points[M])
    ${ /* seq. code */ }$
end


map find_new_crv in(int colour, float a[N], float b[N], float c[N], float d[N])
                 out(colour, float new_y_tpl[N][M])
    find_interm_p in(a[*i], b[*i], c[*i], d[*i])
                  out(new_y_tpl[*i][])
end map
```

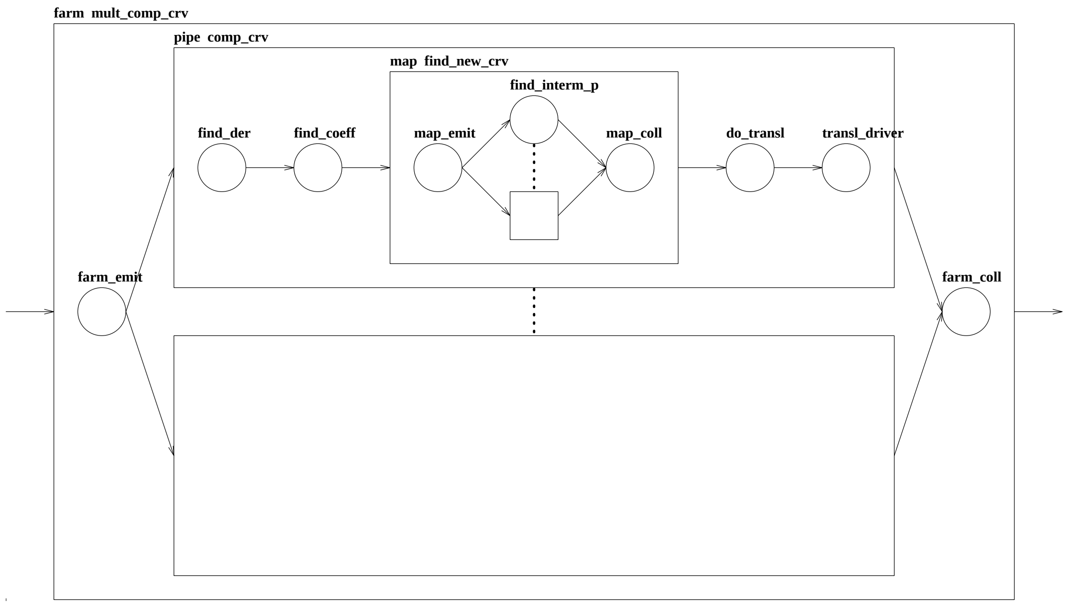


Figure 3: The logical graph of a sample P^3L application.

This example shows how the P^3L constructs can be hierarchically composed. The hierarchical structure of the program is depicted by the *construct tree* that is shown in Figure 2.(b). This tree is extensively used by the P^3L compiler for its optimizations. Figure 3 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 a program.

3 The target architecture, and its abstract machine

P^3L has been designed to easily program massively parallel architectures, which we can identify with the class of the DM-MIMD architectures [22]. Since the class of the k-ary n-cubes [9] includes the most interesting DM-MIMD architectures, e.g. two and three-dimensional meshes, hypercubes, etc.., we refer to this class as the main target of the P^3L compiler.

Within our methodology all the members of the k-ary n-cubes class are viewed through an AM, called P^3M [5]. One of the main feature of P^3M is its ability to subdivide the machine resources, i.e. processing nodes and interconnection network, into two layers. One, the *processing surface*, is used to run processes that can use message-passing to communicate between them, the other, the *data surface*, is employed for implementing an associative *shared-memory*, to which the processes mapped on the processing surface can also access. P^3M exports to the P^3L compiler (1) the topology of the processing surface, (2) a definite mechanism interface [19], e.g. synchronous/asynchronous blocking/unblocking message passing mechanisms, and mechanisms to access a shared memory organized as a tuple space, and, finally, (3) a measure of the costs of the mechanisms exported.

It is worth pointing out that, in our methodology, the mechanisms exported by P^3M are not directly used by P^3L programmers, but are employed by the implementation templates of the P^3L constructs and their compositions. 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. These features are summed up by the mechanism costs exported by P^3M.

The prototype In the rest of the paper we illustrate the design of our prototype compiler, which only produces code for the processing surface of P^3M. More specifically, for this prototype we have considered that the P^3M AM provides a very reduced set of mechanisms: standard sequential operations, process abstraction, and simple message passing between processes allocated to directly connected nodes. We show how this reduced set of mechanisms, similar to those offered by *Transputer-based* DM-MIMD machines, suffices to implement the support of a subset of P^3L, i.e. the *implementation templates* of some P^3L 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 construct are very simple, the choices made by the compiler are more effective;
- simpler and more effective implementations of the mechanisms, as more complex and infrequently-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.

Of the various k-ary n-cubes, the first architectures we have considered as target machines are those which adopt the *two-dimensional mesh* as network topology. The mesh is a particular *low-dimensional* k-ary n-cube network (i.e. the k-ary 2-cube) that is easily realizable, and has been adopted by many of the commercial multicomputer vendors. From the technological point of view, the adoption of the mesh makes a massively parallel architecture highly scalable. To exploit the potential scalability of mesh-based machines, however, it is needed a programming style that exploit locality of computation [6]. As we will see in the following, the programming methodology based upon P^3L can effectively exploit the features of mesh-based architectures.

4 Implementation templates for mesh-based architectures

Several issues must be taken into account to achieve an efficient implementation of a parallel program: they are strongly interrelated, and, each of them, is computationally hard. Some of these issues concern function and data granularity, as well as scheduling and communication strategies to be employed. Suppose, for example, that we adopt a static approach, in which first a static network of communicating process is generated, then this network is mapped onto the actual processor network. Several policies may be embedded within the process network, affecting, for example, the scheduling of the computation units, or the handling of the messages. All these policies may be nullified when this network is mapped onto a real machine. First of all, we have to consider that 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-approximateable* [2, 13]. Moreover, since the mapped processes must interact with the routing sub-system, it is possible that a specific communication pattern, decided during a previous phase, may generate a degradation of the overall performance of the routing sub-system when the mapping choices are actually made.

The solution adopted by P^3L consists in restricting the forms of parallel computation allowed. This restriction leads to the definition of a set of parallel constructs, for which, given a particular AM, it is possible to devise implementations which embodies strategies to solve, at the same time, typical issues like mapping, scheduling, message routing, degree/granularity of parallelism. These implementations, called *implementation templates*, are integral part of the P^3L compiler.

Note that, with this prototype, the aim is to show the use of the templates in our programming methodology. In general, in fact, other, even better implementation templates might be devised for each P^3L construct. These templates, however, should be general enough to efficiently implement the corresponding constructs in every P^3L programs, and should be endowed with accurate performance formulae.

Now we introduce the terminology used in the following. The compiler maintains the knowledge about the implementation templates as a two-level information. One is concerned with the interconnection and the mapping of processes and channels. We refer to them as *mapping template*. The actual code of the processes is maintained as a separated information, within the so-called *process templates*. This distinction is due to the compiler organization, since each of the two kinds of information are used during two successive passes of the compiler.

Note that the implementation templates must not be considered as particular networks of communicating processes. On the other hand, they maintain, in a parametric way, the information about which processes to use, the way in which these processes must be interconnected and mapped, the solution to implementation issues related to the hierarchical composition allowed by P^3L. For example, the template of the `farm` does not fix the parallelism degree to be actually exploited, i.e. the number of parallel workers of `farm`, but it determines the processes to be employed, and the way in which these processes must be interconnected and mapped. The compiler, using the performance formulae associated with the template itself, determines a specific *instance* of the template, i.e. a specific static network of communicating processes where the number of parallel workers has been decided. These formulae are parametric with respect to the costs of the AM mechanisms, where the costs are dependent, in turn, on problem parameters, e.g the size of the data to be transmitted during a given message exchange, or the average number of sequential instructions executed by a sequential function.

In the following of this section we briefly illustrate the templates used in our prototype compiler for mesh-based architectures.

4.1 Process networks implementing the P^3L constructs

We illustrate the implementation templates for a mesh-based architecture by means of directed graphs, which represent the "mapping" of both processes and channels onto a portion of a mesh network (a *sub-mesh*). These graphs are related to the *mapping templates* of the P^3L constructs. In particular, each graph, simply called *mapping*, corresponds to the process network associated with an instance of the template.

Before presenting the templates implementing the P^3L parallel constructs, we illustrate the simple template associated with the `sequential` construct. It is a single process, with an input channel (corresponding to the input parameter list), and an output one (corresponding to the output parameter list). The process is structured as an infinite loop, in which it receives a data item on which a new task must be started, executes the task and assigns the output parameters (this is performed by the sequential code written by the P^3L programmer), and finally sends out the output data item so produced.

4.1.1 The `farm` construct

The logical structure of the `farm` shown in Figure 2.(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, which sends out the output data stream of the `farm` construct.

In the actual implementation, further requirements have been taken into account. In fact, the distribution activity has to perform the dynamic load-balancing between the workers (due to the variable completion times of the tasks), while the collection activity has to reorder the task results.

The logical structure of Figure 2.(a) cannot be directly mapped onto a mesh. In fact, both fan-in and fan-out of some processes may be greater than four, i.e. the node degree of the mesh topology. Thus, the implementation template adopts a solution in which the activities of the `emitter` and the `collector` are distributed to rows of processes. Moreover, the distribution of

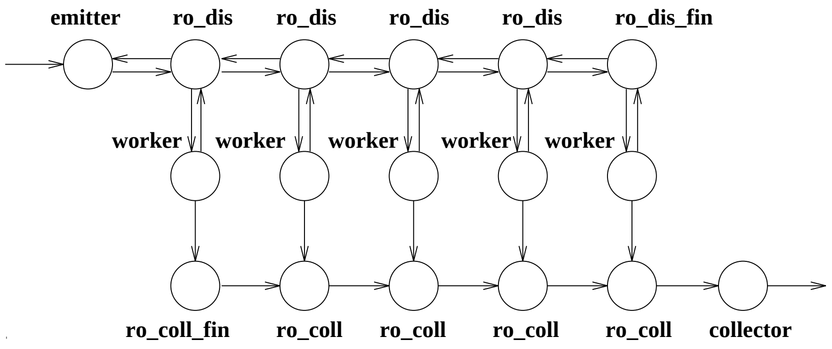


Figure 4: A mapping of a `farm`.

the `emitter` activity prevents the slow-down due to the, otherwise centralized, scheduling policy. Figure 4 shows a particular instance of the mapping template of the `farm` construct, characterized by five copies of the nested sequential worker. The task distribution activity is implemented by the processes `emitter` (emitter process), `ro_dis` (distributing router process) and `ro_dis_fin` (final distributing router process). The result collection activity is performed by the processes `collector` (collector process), `ro_coll` (collector router process) and `ro_coll_fin` (final collector router process). The details of this implementation template for the `farm` are reported in Appendix A.

Note that some *general purpose* processing nodes are exclusively used to route and schedule data packets, and to balance, at the same time, the computational load between the workers. This template is very effective, and, for its generality, has been included in the prototype compiler. This, however, is one of the possible implementations of the farm on an AM without hardware routing layer. For example, other implementations wasting less resources have been studied. These other implementations could be used when the resources are limited, a smaller parallelism degree must be exploited, and, thus, a very high distribution/collection bandwidth is not longer needed.

4.1.2 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:

$$\begin{array}{c} [in_1, \ldots, in_n] \\ \downarrow \\ [[in_1^1, \ldots, in_1^m], \ldots, [in_n^1, \ldots, in_n^m]] \\ \downarrow \\ \mathcal{M} \\ \downarrow \\ [[out_1^1, \ldots, out_1^m], \ldots, [out_n^1, \ldots, out_n^m]] \\ \downarrow \\ [out_1, \ldots, out_n] \end{array}$$

where $\mathcal{M}$ represents the module corresponding to the nested construct of the `map` (i.e. a *worker* of the `map`), 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.

The difference between the `farm` and the `map` templates concerns the processes `emitter` and `collector`, which 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.

4.1.3 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 5 shows the mapping of a `pipe` construct composed of nine sequential stages. A heuristic is used by the compiler to map all the stages of the `pipe` on a regular sub-mesh, allocating the processes that belong to distinct stages and need to communicate - i.e. the processes that produce the output stream for each stage, and the processes that receive the input stream for the next stages - as close as possible (e.g. on neighboring processing nodes). Sometimes, special routing processes are allocated to implement non-local communications.

4.1.4 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 6, which shows the logical structure and the mapping of a `loop` construct. The `in_l` process implements the merging between the tasks flowing along the `input` and the `feedback` channel (which is implemented as a chain of router processes). The `out_l` process receives the results from the nested module, and checks whether the final condition has been reached. If the final condition is false, it generates a new task to be sent back to the `in_l` process along the `feedback` channel. Otherwise, i.e. if the condition is true, it returns the result of the nested module along the `output` channel. The ordering of the stream is maintained by using an ordered structure, in the same way as 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.

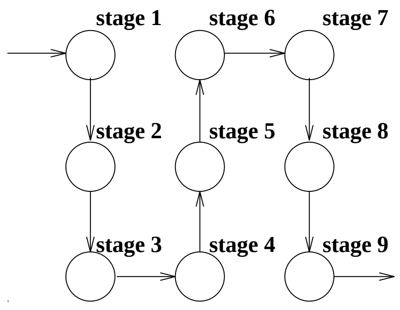


Figure 5: A mapping of a `pipe` construct composed of nine sequential stages.

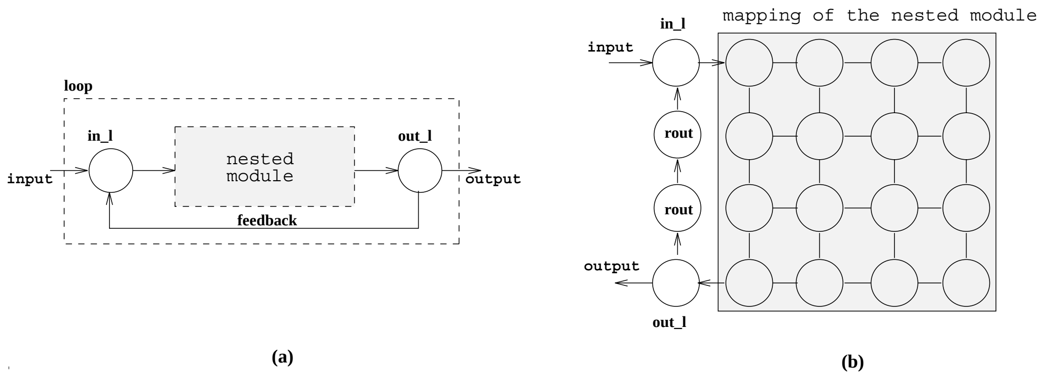


Figure 6: A `loop` construct: (a) logical structure, and (b) mapping.

4.1.5 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 P^3L `sequential` constructs check whether the received input data items are *end-of-stream* packets. If it is, the packets are by-passed, and are 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 this section we ilustrate the solution adopted by our prototype compiler to map P^3L program that comprise several, hierarchical composed, constructs on a mesh. The compiler associates with the mapping of a given construct a 2-dimensional box, with a pair of input/output channels. This box is the smallest one that encloses the graph. If the compiler has to deal with architectures characterized by different interconnection network topologies, the dimension number of the boxes is different. For example, if the architecture belongs to the class of the k-ary 3-cubes, we have to adopt *three-dimensional boxes.*

The box abstraction is used by the compiler to maintain, in a parametric way, the knowledge

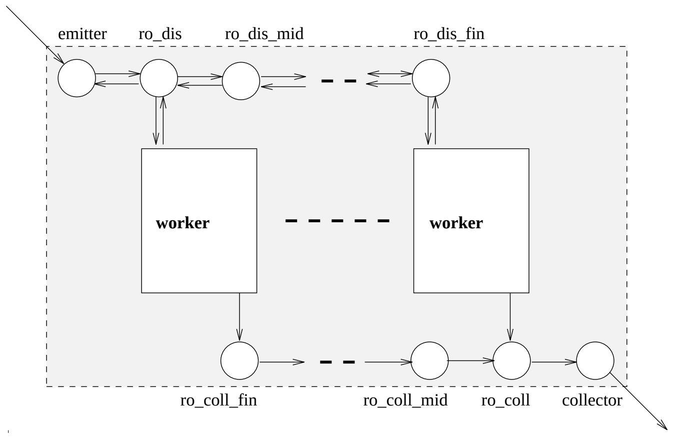


Figure 7: A *mapping template* of the `farm`.

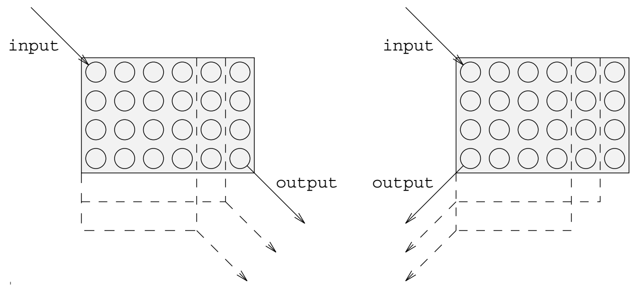


Figure 8: Possible *boxes* enclosing mapping templates.

on each mapping template. In fact, the parameters of each mapping template are exactly the features of the boxes corresponding to nested constructs. For example, Figure 4 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 7, 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.

The current prototype compiler exploits boxes with some restrictions. Figure 8 shows the boxes allowed. 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 (i.e. the processes and the channels contained in the box).

The next section shows how several mapping templates must be supplied to the compiler for each P^3L construct. These mapping templates are different in the features of the enclosing box. For example, another general mapping template for the `farm` construct is included in the compiler. This other template can be obtained by changing the orientation of the collection chain in the mapping template in Figure 7. In this new template 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

As previously stated, the compiler tunes the implementation of each construct by using the analytical performance formulae associated with each template. Our prototype compiler uses some performance models that allows the compiler to determine the best parallelism degree in order to minimize the *service time* (or, equivalently, to maximize the *speedup*) of each implementation in computing a stream of tasks [21]. This approach permits the compiler to allocate statically all the resources needed to run a given program. We are also devising some heuristics that, when the number of processing node is limited, reduce the parallelism exploited in the implementation of a P^3L program in order to achieve the best service time and the best resource utilization [3].

In Appendix B we illustrate in more detail the *analytical performance models* associated with the implementation template of the `farm`, and briefly discuss the models associated with the others. The models depend on a set of parameters relative to the costs of the P^3M mechanisms, and the performance of the nested construct(s). Some examples of the P^3M mechanism costs are the time taken by communications, or the time taken by certain processes produced by the compiler (e.g. `emitter`, `collector`, etc.) to perform some particular actions, such as buffer copies, flag check, etc. Some of these costs are supplied as a function of the problem data size. Other important costs are concerned with the average time spent by user-provided sequential code. The kwoledge of these costs is necessary when `sequential` constructs are nested in other parallel constructs. Currently, our prototype derives the average time taken to execute a `sequential` construct by profiling an implementation of the P^3L program. However, we are devising a static tool that, on the basis of the average time spent to compute the sequential instructions of an intermediate language (related to the specific AM), computes the average execution time of a fragment of sequential code. If the tool fails to obtain this cost statically, it returns some dependencies of this cost from the distribution of the input data. This information is used to select suitable data samples for the profiling of the sequential code.

5 The structure of the compiler

In the previous section, we have 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 *mapping library*, which contains the *mapping templates* of each construct along with the *analytic performance formulae* associated with them. The *mapping library* is accessed by the compiler through a set of *rules*. All these rules are arranged in another library, called the *optimization library*.

Finally, the compiler access another library called *process template library*. It contains the code of the sequential processes, i.e. the *process templates*, which are referred by the mapping templates included in the *mapping library*. For example, the `emitter` and the `collector` are two of the processes included in the *process template library*. To discuss in more detail how all these libraries are actually exploited, it is useful to refer to the general design of the P^3L compiler, which is composed of the three following passes:

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 2.(b). Each node of the tree is annotated with information about the parameter passing between the hierarchically composed constructs, and, for the `sequential` constructs, with the files and of the procedures containing the user-provided code.

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 types of the data flowing over the channels, and the mapping of processes and channels onto an abstract representation of the target architecture. To accomplish this task, it accesses the *mapping library* through the 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 data structures produced by the middle-end, generates the actual code for P^3M. This task is carried out by using the predefined process templates included in the *process template library*.

The general design of the P^3L compiler is shown in Figure 9, which also illustrates the interactions of each part of the compiler with the libraries and the P^3M.

5.1 The libraries

Below we describe in more detail the libraries, assuming that they all relate to an AM exporting a given network topology.

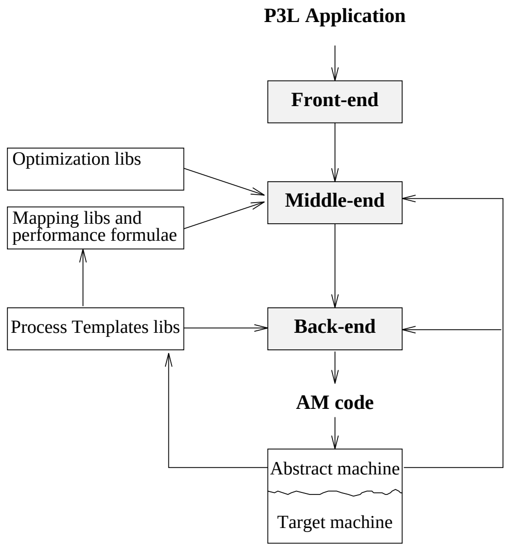


Figure 9: The structure of the P^3L compiler.

Mapping library The *mapping library* contains several 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 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 process network implementing the nested construct of $\mathcal{C}$ (in case $\mathcal{C}$=`pipe`, since the `pipe` has several nested constructs, more of these structures are included in $\mathcal{N}$). As discussed above, at this level the nested constructs are thought of as boxes 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 that will have to enclose the process structure implementing the construct $\mathcal{C}$.

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

- $\mathcal{N}'$ is the data structure representing the optimized process network implementing $\mathcal{C}$. The performance formulae associated with the specific library entry are used to optimize $\mathcal{N}'$. Note that the same $\mathcal{N}'$ 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.

Both $\mathcal{N}$ and $\mathcal{N}'$ are data structures of the same type as those generated by the *middle-end* as final output.

Process template library The code corresponding to each process template is written by using the *host sequential language*, plus the concurrency mechanisms exported by P^3M. Basically, this code refers to a set of channels, which are used by the communication mechanisms, and a set of data structures, which corresponds to the data types transmitted over 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 process 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, or further parallelism can be exploited by introducing other parallel constructs. The two kinds of rules included in the *optimization library* can be formally described by

$$\mathbf{R1}: \quad \frac{\mathcal{P} \qquad \mathcal{M}}{\mathcal{O}\mathrm{pt}_{\mathrm{Map}}}$$

$$\mathbf{R2}: \quad \frac{\mathcal{P} \qquad \mathcal{M}}{\mathcal{O}pt_{Tree}}$$

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. 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, performs the type checking on the input/output parameters lists of each P^3L construct involved, and produces the construct tree of the program, which describes the hierarchical composition of the P^3L constructs. In addition, the fragments of C++ code provided by the programmer for each `sequential` construct (enclosed in `${ }$`) are included in C++ `void` functions, stored in special files. These files are separately compiled using the host language compiler to check their syntactical correctness.

The *front-end* has been realized by using the standard `lex` and `yacc` tools. The construct tree is exported as a set of Prolog data structures. In fact, for easy prototyping, all the other parts of the compiler have been written in Prolog.

Figure 10 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 10), or the host language files including the user-provided code (this only occurs for the `sequential` constructs, e.g. the Prolog fact `seq(..)` in Figure 10).

```
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
```

$$\xrightarrow{front-end}$$

```
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', 'p31__0__', './p31__0__.C',
  src(['m.C']),
  libs([])).

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

Figure 10: Example of parsing a simple P^3L program.

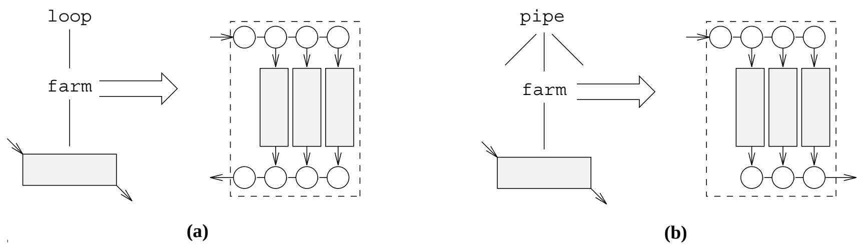


Figure 11: A mapping of a `farm` depending on the type of its parent, (a) `loop`, or (b) `pipe`.

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++ `void` function including the user-provided code. The third field identifies the file that includes the C++ function. 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 other Prolog structures, namely the *mapping data structures*. The algorithm adopted is based on a *depth-first visit* of the construct tree. The 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*. For each entry selected, the library returns a given process network, represented by a *mapping data structure*. In some cases, instead of selecting a *mapping library* entry, the rule employed modifies the construct tree.

Now we show how the *middle-end* works by means of an example. Suppose that, during the visit of the construct tree, a `farm` is encountered. Since the visit is depth-first, we can assume that, at this point, the process network implementing the descendants of the `farm` has already

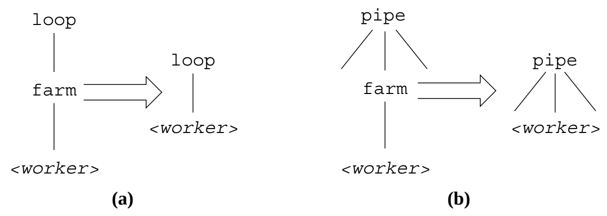


Figure 12: Tree transformations involving a `farm`, whose parent is (a) a `loop`, or (b) a `pipe`.

been generated, and is enclosed in a given box. It is now useful to consider two different constructs in which the `farm` can be nested: a `pipe` and a `loop`. Figure 11 illustrates a portions of the two trees, where, in place of the worker of the `farm`, the associated box is shown.

The *middle-end* uses the rules (R1), included in the *optimization library*, to select a given mapping template for the two `farm` constructs. The rules applied have, of course, a distict matching precondition $\mathcal{M}$, i.e. either `loop(farm)`, or `pipe(farm)`. Moreover, since the $\mathcal{P}$ preconditions are concerned with the bandwidth of the nested construct implementation, in both the rules $\mathcal{P}$ holds if and only if the bandwidth of the worker is *low enough* to make it useful to exploit `farm` parallelism. Finally, each of the two rules selects a different entry of the mapping libraries through its own $\mathcal{O}pt_{Map}$ action. The final process network is shown on the right of the two trees in Figure 11. Note the different features of the boxes enclosing the final process networks. For example, the box of Figure 11.(a) has the input and output channels placed on the same side of the box. In fact, if we consider the mapping template of the `loop` construct (shown in Figure 6), we can understand that 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.

If no rules (R1) can be applied, the middle-end algorithm tries to apply a rules (R2). As regards our example, the rules (R2) can be selected only if the bandwidth of the worker of the `farm` is very high, so that the `farm` parallelism can be considered as not useful. Thus, the $\mathcal{P}$ preconditions are the negation of those appearing in the rules (R1). The $\mathcal{O}pt_{Tree}$ action of the selected rule removes the `farm` construct from the tree, preserving, at the same time, the semantics of the program. Figure 12 shows the tree-to-tree transformations for both the examples in Figure 11.

Mapping data-structures produced by the middle-end During the visit of the tree, each time a given construct is encountered, the *middle-end* generates some data structures, called *mapping data structures*, representing a process network implementing that construct. A very small subset of these structures specifies the mapping of processes and channels, while the other structures, whose meaning is discussed below, are bound with this subset by means of special identifiers. Flippings and rotations that may be needed to map the various constructs only change this small subset of data-structures. 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 needs to be replicated.

Consider the simple program shown in Figure 10. The *middle-end* receives the tree representation of the program, and produces the process network shown in Figure 13. Figure 14 shows a small subset of these data structures, describing the mapping of the process network. It consists of a `map(..)` Prolog fact, which contains a list whose elements are, in turn, `map_real(..)` structures describing the mapping of a single process along with its channels:

```
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

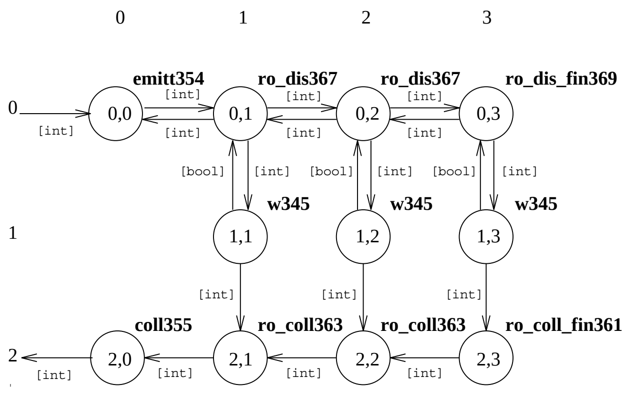


Figure 13: A mapped process network implementing a P^3L program.

Ind2 are two numerical indexes identifying a given processing node of a 2-dimensional mesh[1]. The Templ_id key is used to identify all the processes characterized by the same code and the same channel types. 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 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 initial distribution of the tasks. The fourth and the fifth components are two lists, describing the channels used to receive and send messages, respectively, from neighbouring nodes. 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, a Type_id is associated with each channel, identifying the type of the data to be sent/received over the channel.

Besides the data structure presented above, the *middle-end* generates other data structures, which complete the set of the *mapping data structures*. For example, there are Prolog facts that, for each Type_id, specify the type of the data to be transmitted over the channels, while there are others that specify the process template corresponding to each Templ_id key. Note that, for each process template included in the corresponding library, we can have several different processes, each identified by a distinct Templ_id. For example, two emitter processes implementing two farm constructs are not identified by the same Templ_id's, even if they are implemented by using the same process template. In fact, the types of the channels and the data structured handled by the two processes are not the same, in general, but depend on the input parameter list of both the farm construct and its nested worker.

5.4 The back-end

The last part of the compiler is the *back-end*, which heavily depends on the interface provided by P^3M. 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 P^3M, as well as the actual code for the process (with the relative channels) to be mapped on each processing node. Therefore, one of the main activities performed by the *back-end* consists in translating from the *mapping data structures* generated by the middle-end into the AM configuration language of P^3M. Another other important task deals with the production of the actual code for each process, determining the host-language modules and libraries to be linked together. The *process template library* provides this code that, however, needs to be completed adding the

[1]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.

```
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), process(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), process(coll355), param([]),
         [receive(proc(2, 1), coll_bus_in, type353)],
         [send(proc(2, -1), out, type350)]),

      map_real(proc(0, 1), process(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 14: An example of the data structures generated by the *middle-end.*

definitions of the specific data structures and the macros used inside this code.

Currently, the *back-end* produce code for an emulator of parallel architectures, and for a real parallel machine, a Meiko CS1 based on T800 Transputer.

6 Related work

The purpose of our research is to make the parallel architectures easily programmable, and at the same time, to supply tools to exploit the specific features of each target architecture, maximizing, for example, the "useful" parallelism of each application. Thus, due to its generality, we can identify several research tracks that share this goal with us.

A research track related with our work is the so-called *universal models for parallel computation* proposed by Valiant [25]. In this case, the goal is very ambitious. It consists in finding a computational model that can be simulated onto different target architecture with a bounded and predictable slow-down, thus keeping constant the product between the number of processors used and the total execution time. In this way, programmers can develop their programs in a machine independent way, and can evaluate the performance of their programs without running them. Unfortunately, as pointed out by Bilardi and Preparata [6], the architectures that seem to be more promising for the future parallel systems, like *low-dimensional* k-ary n-cubes, are exactly those for which it is more difficult to simulate such universal models. Our approach is more pragmatical, in the sense that we are interested in developing powerful and effective tools especially for those low-dimensional "difficult" architectures.

Our approach appears more related to the work in progress on the efficient massively parallel implementation of functional languages based on a *restricted set of second order functionals* [8, 12, 23]. In this case the idea consists of encapsulating certain common algorithmic forms in higher-order functions to facilitate parallel program development.

In particular, Cole [8] and Darlington et al. [12] propose systems in which the user is provided with a *library of skeletons.* Each skeleton is implemented by a set of templates on different target architectures, but the program restructuring, needed to match different hardware requirements, is totally in charge of programmers.

The approach proposed by Skillicorn [23] addresses the issue of composing paradigms, but it only considers data parallel computations, restricting the control flow to a single thread expressed as a sequence of function compositions. Since the approach is based on strong and clear semantic properties, it also allows program transformation techniques to be devised, and allows a program implementation to be derived from an initial high-level specification using algebraic identities between higher order functionals.

Finally, we discuss some relationships with the approach followed by researchers working on Fortran dialects for programming DM-MIMD machines [16, 15]. Considering the features of High Performance Fortran (HPF), we can deduce that HPF allows parallel programs to be expressed in a sufficiently machine-independent way. Moreover, its `FORALL` construct and some of its intrinsic functions resemble some P^3L constructs (or compositions of some P^3L constructs). The purpose of HPF is, however, to permit a programmer to easily express *data-parallel* computations, while P^3L also covers *control parallel* ones. Another difference is related to the role of the compiler, which, in our methodology, uses a cost model to predict the performance and, thus, to restructure the program implementations to harness each specific target architecture. Therefore, the lacking of a cost model for HPF implementations may prevent several compiling optimizations allowed by P^3L, and may make it difficult to adopt a solution in which HPF is used as P^3L host language.

Finally let us assess our methodology with respect to the coordination languages and the corresponding computational models. *Linda* [1], which is one of the most important examples of this kind of languages, 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. Moreover, the Linda supports may incur in high overheads due to the emulation of an unrestrictedly accessed shared address space.

The same lack of performance portability is, in our opinion, the main drawback of high-level programming environments based on graphical development tools. In fact, even if they provide a friendly interface for parallel systems running libraries and run-time software such as PVM [24], performance tuning is always the responsibility of the programmer.

7 Conclusions and future work

The prototype of the P^3L compiler described in this paper has been developed during a joint project, called P4 [4], 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 has been implemented on top of an emulator of parallel architectures, i.e. Proteus [7, 20, 18], 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. Moreover, we have just completed the porting of the whole programming environment, and thus also of the AM without the shared address space, to an actual machine, the Meiko CS1, a DM-MIMD machine based on T800 Transputers.

Other current activities are concerned with devising implementation templates, performance models, and process templates for the parallel P^3L constructs that haven't been included yet [11]. These other constructs model

- `reduce` computation, which 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 of type $\mathcal{T}$;
- `tree` computations, of which a computation built of a `map` followed by a `reduce` is a particular case;

- `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 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.
- `dedicated-farm` computations, where distinct functions are provided, but, on the basis of distinct *guards*, only one is selected for each item of the input data stream.
- `divide-et-conquer` computations, where each item of the input data stream is partitioned by using a *dividing function*, a *computing function* is executed in parallel on each partition producing many parallel results, and, finally, a *combining function* is applied on these results to recompose an item of the output data stream.

Further activities on P^3L are concerned with porting the language and its environment to architectures with network topologies different 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.

Acknowledgements

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 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. We also thank Fabio Piazzai and Francesco Chiaravalloti for porting the P^3L environment to the Meiko. We are grateful to the Hewlett Packard Laboratories, which supported this research, and all the P4 team at the HP Pisa Science Center. Finally, we thank the anonymous reviewers for many useful comments and suggestions.

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 Science*, 3(3):379–387, 1993.

[3] B. Bacci, M. Danelutto, and S. Pelagatti. Resource optimization via structured parallel programming. Technical Report TR-30/93, Dipartimento di Informatica, 1993.

[4] 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*, pages 536–543, Tokyo, Japan, 1991.

[5] F. Baiardi and M. Jazayeri. P^3M: a Virtual Machine Approach to Massively Parallel Computation. In *Int. Conf. on Parallel Processing*, August 1993.

[6] G. Bilardi and F.P. Preparata. Horizons of Parallel Computation. Technical Report CS-93-20, Dept. of Computer Science - Brown University, May 1993.

[7] E.A. Brewer and C.N. Dellarocas. Proteus User Documentation Version 0.2. Technical report, Massachusetts Institute of Technology, October 1991.

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

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

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

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

[12] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *Proc. of PARLE '93 - 5th Int. PARLE Conf.*, pages 146–160, Munich, Germany, June 1993. LNCS 694 Spinger-Verlag.

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

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

[15] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[16] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):67–80, August 1992.

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

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

[19] 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.

[20] 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.

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

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

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

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

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

Appendix A: The implementation of the `farm`

This appendix illustrates in detail the three activities, namely task distribution, computation, and result collection, which we can recognize in the `farm` implementation template. A particular mapping of this template is shown in Figure 4.

Implementation of the task distribution The `emitter`, `ro_dis` and `ro_dis_fin` realize the distribution of the tasks by implementing a *slotted ring*. 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 any of the tasks, but they are *tagged* with an increasing mark (*tag*) to maintain the input/output ordering. Figure 15 shows the ring in more detail, where the black circles in each process represent the slots. The tasks moves on the ring according to the directions of the channels.

At the steady state, all the processes that implement the ring communicate simultaneously with their neighbours, so that the contents of the slots are moved by a single position. After, if the `emitter` process holds an *empty* slot, it copies an incoming data item (from the input stream) to this slot. Moreover, if if the `ro_dis` and `ro_dis_fin` processes 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 by selecting first the data item associated with the oldest tag.

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`), (2) sends a new request for further tasks, (3) executes the task (the sequential code written by the P^3L programmer), and (4) sends the result of the computation to the corresponding `ro_coll` (or `ro_coll_fin`).

Note the `synch` channel shown in Figure 15, used to request a new task. 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.

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

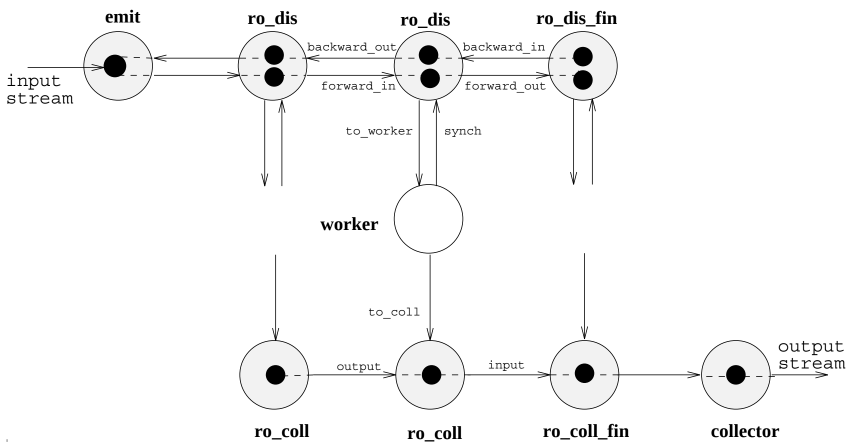


Figure 15: Implementation of the task distribution and result collection.

empty or may contain a result to deliver to the `collector` process. Figure 15 shows these processes and the channels between them. The black circles represent the slots. The results move on the chain following the direction of the channels.

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.

Appendix B: The analytical performance models

In this appendix we introduce the *analytical performance models* which are associated with each implementation template, and which allow synthetic formulae to be derived and used by the compiler. In particular, we describe in more detail the analytical model used to derive the performance formulae associated with the `farm`. These formulae are used to devise (1) the useful parallelism degree (i.e. the number of workers in the `farm`), and (2) to return the bandwidth of the final implementation.

As discussed in Appendix A, in the implementation of the `farm` we can distinguish three pipelined stages, i.e. the *distribution*, the *computation*, and the *collection* stage. In order to maximize the bandwidth, we have to make faster the stages that appear to be the bottlenecks of this pipeline. When the bandwidths of the three stages are balanced, to improve the total bandwidth we have to make faster all the three stages. If this is impossible for only one of the stages, we can deduce that the best total bandwidth has been achieved. 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).

In this case, 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. τ_p can be express as a function of the AM mechanism cost, such as the time spent to copy a data item in an empty slot, to test some flags, and to perform message exchanges between neighbouring processing nodes. In addition, the bandwidth of n_w workers is $n_w\beta_w$, where 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 $\beta_w = \frac{1}{\tau_w}$.

To balance the bandwidth of the three functional stages of this `farm` implementation, we have to find an n_w such that $\frac{1}{\tau_p} = n_w\beta_w$, which holds iff

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

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 performs the dynamic scheduling and guarantees an optimal load balancing even if there is a very large variance of τ_w. An example of this feature

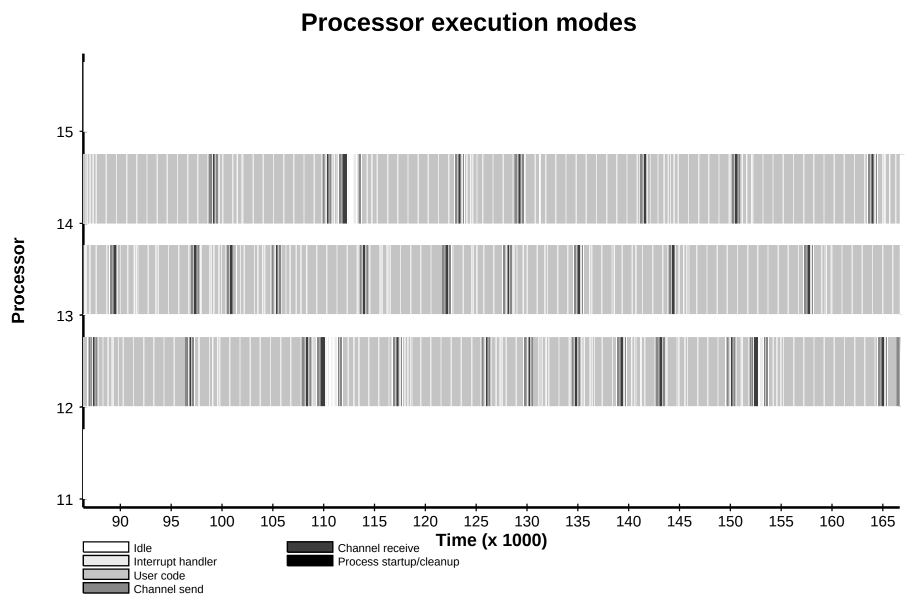


Figure 16: Example of load balancing obtained by a `farm` implementation.

is shown in Figure 16, which illustrates the behaviour 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 [20], on top of which P^3Mhas 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 number of workers has been found by using (1) w.r.t. to the average value of τ_w, i.e. the midpoint of the distribution interval.

Similar formulae are used by the compiler for the other P^3L constructs. The `map` formulae are the same as the ones 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 stages, while the final bandwidth is the smallest 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 `in_l` and in the final condition check performed by `out_l`.