Rewriting skeleton programs: how to evaluate the data-parallel stream-parallel tradeoff

M. Aldinucci, M. Coppola & M. Danelutto

Department of Computer Science – University of Pisa
Corso Italia, 40 – 56125 Pisa – Italy
Email: {aldinuc,coppola,marcod}@di.unipi.it

Abstract. Some skeleton based parallel programming models allow the programmer to use both data and stream parallel skeletons within the same program.
It is known that particular skeleton nestings can be formally rewritten into different nestings that preserve the functional semantics. Indeed, the kind and possibly the amount of parallelism usefully exploitable may change while rewriting takes place.
Here we discuss an original framework allowing the user (and/or the compiling tools) of a skeleton based parallel programming language to evaluate whether or not the transformation of a skeleton program is worthwhile in terms of the final program performance. We address, in particular, the evaluation of transformations exchanging data parallel and stream parallel skeleton subtrees.

1 Introduction

Since the work of Cole on skeletons [1], many researchers worked out the idea of providing the parallel programmer with a set of skeletons that could be used to model the parallel structure of applications and algorithms. Different research teams have designed skeleton sets that vary in the number and specialisation of the skeletons, as well as in the possibility of performing arbitrary skeleton nestings within programs [2, 3].

The skeleton idea is very appealing mostly due to the fact that skeletons allow efficient development of parallel applications by proper specification of the skeleton nesting to be used. All the details of process decomposition, mapping and scheduling, as well as the communication and memory management issues are handled by the skeleton system in place of the programmer. This pleasant property is paid in two ways. On the one hand, programmers are not allowed to exploit any arbitrary parallel decompositions of a problem: they are forced to use just those patterns provided by the skeleton system (skeleton models belong to the "restricted programming model" class [4]). On the other hand, the implementation of an efficient skeleton based programming environment is clearly a hard task, as all the process and communication details the programmer has not to deal with must be handled by the skeleton system designer.

In most skeleton frameworks equivalences can be established between different skeleton nestings. We can state that:

$$E_i : \; X_1(X_{11} \dots X_{1n}) \equiv X_2(X_{21} \dots X_{2m})$$

meaning that skeleton X_1 applied to skeletons X_{11} to X_{1n} *computes the same result* of skeleton X_2 applied to skeletons X_{21} to X_{2m}. Equivalences such as this one are of interest to the skeleton programmer, as he can use them as rewriting rules to rewrite his programs in order to achieve a better program structure *and/or* a program sporting a better performance. These equivalences are also of interest to the designer of the skeleton system/language, as he can use them to improve the compilation process, once an algorithm is available stating which equivalences have to be exploited for better performance (efficiency). In both cases, it is fundamental to devise a framework that could be used to evaluate the impact of an equivalence rule over a given program. In other words, it is fundamental to have a formal way of understanding, given two skeleton programs P_1 and P_2, such that P_2 has been obtained transforming P_1 by using a set of equivalences $\{E_1, \dots, E_k\}$, which one is the "better" in terms of the performance achieved (it is supposed to achieve).

In order to be able to answer such kind of questions, we must be able to solve different problems:

- we must be able to enumerate the programs that are "equivalent" to a given program modulo a set of rewriting rules.
- we need a reliable algorithm to compute the expected performance of a skeleton program onto a target machine
- we need an evaluation criteria able to discriminate, among equivalent programs, the one(s) exhibiting the best performance

These problems are not easily dealt with. In this paper we want to investigate the possibility of providing the user with some formal reasoning methodology allowing him to understand whether a given rewrite rule can be efficiently applied to his programs. Furthermore, since the skeleton system we will take into account will contain skeletons exploiting both data and stream parallelism, we will focus on the rules that transform either kind of parallelism into the other. Finding the best interplay among them with respect to the program performance will be our goal.

In the following we will at first show our skeleton model, which is a subset of the P3L one, and a sample of its rewriting rules. We will then briefly outline the design of a performance model for skeleton implementations on real target machines, thus providing a formal way to "measure" the expected performance of left and right hand sides of a rewriting rule. The performance model is built using a *logP*-like cost calculus [5] that extends similar ones already developed, in particular within the P3L project [6, 7]. We will finally show how such a model can be used to tell the programmer what rewriting rule to use and in which direction (left hand to right hand side or vice-versa), depending on some parameters relative to the data, the program structure and the target architecture at hand.

2 The skeleton framework

The skeleton framework we take into account in this work is a subset of P3L. P3L is a parallel programming language based on skeletons, under development at the University of Pisa since 1991 [2, 7, 6]. All the skeletons in P3L can be nested. The semantics of P3L skeletons is data-flow like, as they all process an input data stream, producing as a result an output data stream whose elements are computed by applying some function over the items of the input stream (we will often refer to single data items belonging to the streams as *task*s).

Both stream and data parallel skeletons are provided: even the data-parallel skeletons work on streams, in that they process all the items appearing onto the input stream, but they are meant to exploit the data parallelism within the computation of a single input task. The basic skeleton (seq) encapsulates computations expressed in a sequential language. Of course it is an exception as it cannot contain any other skeleton. Iterative (data and stream) parallel computations are handled by a loop skeleton, but we will not deal with it in this paper.

Stream parallel skeletons Concerning stream parallelism, a farm, a pipe and a loop skeleton are made available to the programmer. The farm skeleton computes a given function over the input tasks by using a set of identical, parallel worker processes. If f is the function computed by the nested skeleton X, for each data item d in input the structure $\mathsf{farm}(X)$ produces onto the output stream the item $f(d)$. The pipe skeleton computes the composition of a given number of skeletons on each task d of the input stream. If f_i is the function computed by the skeleton X_i, $\mathsf{pipe}(X_1, \ldots, X_k)$ computes $f_k(f_{k-1}(\ldots f_1(d)\ldots))$. Parallelism is exploited by setting up an independent, parallel process for each pipeline stage. The amount of parallelism in this skeleton is fixed by the programmer, as we will not deal here with the stage-merging optimizations explained in [8].

Data parallel skeletons The data-parallel skeletons comp, map and reduce are provided to the user. The comp skeleton has the semantics of the functional composition, just like the pipe skeleton, but it does not deal with stream parallelism. Instead it uses a unique set of independent processing elements to sequentially evaluate the functions of the skeleton expressions given as stages of the comp. Since various skeleton expressions "overlap", each single processing element sequentially participates in the computation of each stage, possibly taking part in collective communication operations to redistribute data between stages. We want to stress the point that no parallelism is introduced by the comp itself: data parallelism is exploited by the nested skeletons. We just mention the fact that iterative data-parallel computations can be expressed using a combination of the comp and loop skeletons.

The $\mathsf{map}(X)$ skeleton applies the function f computed by the nested skeleton X to all the items of a set of subsets of the input data structure. Each application of f over a single independent data subset is usually referred to as the map *virtual processor* computation (we will speak of map virtual processors meaning

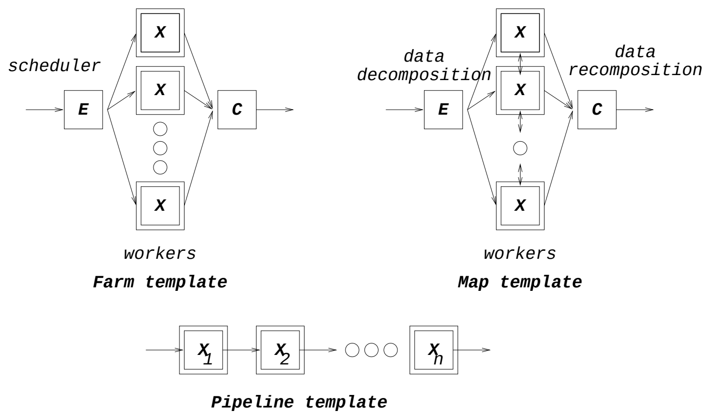

Fig. 1. Implementation templates for the P3L skeletons considered here

these "minimum grain" parallel computations). A set of identical, parallel worker processes applies the function to the data subsets. The user specifies through the map skeleton syntax how the subsets can be computed out of the input data, what function is to be computed on them and how the results can be "merged" to build output data structures.

The reduce(X) skeleton folds all the sub-items of an input data structure using the binary, associative and commutative function computed by the nested skeleton X.

Implementation templates When a P3L program is compiled, each skeleton is carried out using an *implementation template*, i.e. a known, efficient and parametric way of implementing a skeleton onto a given target architecture [7, 9]. Implementation templates are stored in libraries along with performance models. The result of the compilation is a process network implementing the skeleton program semantics and running on the target parallel architecture. The implementation templates used within P3L are outlined in Figure 1. The double boxes represent nested implementation templates (possibly even seq processes computing user sequential code) and simple boxes represent system generated processes. Arrows represent communication channels (either true communications or shared memory accesses) directed according to the program data flow. In our farm template, data items are scheduled on-demand to worker processes by a scheduling process (E), and output data items are delivered onto the output stream by another process (C). The map template is similar, but for the fact that the scheduler process E performs the decomposition of the input data, and the process C performs recomposition of the output structure out of the sub-tasks. The comp template is not shown as it is basically the sequential execution of the nested templates onto a unique set of processing elements.

Most of the implementation templates used within P3L are parametric in the parallelism degree supported; the parallelism exploited is fixed at compile-time, not in the source program. The exact value for the template parameters can be derived in such a way that a good performance is achieved by the resulting process network. For instance, the number of workers in the map can be computed in such a way that the processing bandwidth of the workers matches with the task delivering capacity of process `E` and with the task receiving capacity of `C`. In order to automatically compute these figures, basic parameters must be known, such as the `E` and `C` processes comunication bandwidth and the data decomposition overhead, as well as the average time spent in computation by the worker processes. In order to obtain the data needed to perform this kind of calculations, some performance models such as the one discussed in Section 4 have been developed within the P3L project.

3 Rewriting rules

Within the skeleton framework outlined in Section 2, many rewriting rules stating equivalences between skeleton trees can be established. As an example, at any point where we have a skeleton tree X we can insert a farm, thus obtaining the skeleton tree $\mathsf{farm}(X)$. This because the farm skeleton just enhances the parallelism degree of a computation without affecting the function computed. Therefore the following rule (*farm introduction/elmination* rule) holds:

$$X \equiv \mathsf{farm}(X) \tag{1}$$

Like the other rewriting rules discussed in this Section, the rule (1) is to be intended as a couple of rewriting rules: the left-to-right rule ($(1^{\rightarrow})$, for short), namely the rule $X \rightarrow \mathsf{farm}(X)$ and the right-to-left rule ($(1^{\leftarrow})$, for short), namely $\mathsf{farm}(X) \rightarrow X$. In this work, we will always refer to right-hand and left-hand side with respect to the bi-directional rule, even when speaking of the two derived rules with $\rightarrow$ in place of the $\equiv$.

Rules such as rule (1) hold independently of the parameters of the skeletons involved in the transformation. There are also rules that only hold if certain conditions over the parameters of the involved skeletons hold. As an example, consider the following rule:

$$\mathsf{map}(\mathsf{pipe}(X_1, \ldots, X_k)) \equiv \mathsf{pipe}(\mathsf{map}(X_1), \ldots, \mathsf{map}(X_k)) \tag{2}$$

This rule corresponds to the well known "map distributivity" of the Bird-Meertens formalism [10] ($\mathsf{map}(f_1 \circ \ldots \circ f_k) \equiv \mathsf{map}(f_1) \circ \ldots \circ \mathsf{map}(f_k)$) as well as to the "map fusion law" of *SCL* [3]. In the P3L framework, the rule states that we can exchange the pipe skeleton with the map one. The rule $(2^{\rightarrow})$ always holds, whereas rule $(2^{\leftarrow})$ only holds if all the maps of the right hand side use the same task decomposition (i.e. if the data decomposition performed by the different maps in the right-hand-side of rule (2) in order to obtain the finest task decomposition is the same).

A large set of rules holding in the P3L framework can be found in [11]. Beside the farm rule stated above, the following rules, which are of interest for this work:

$$\mathsf{map}(\mathsf{pipe}(X_1,\ldots,X_k)) \equiv \mathsf{pipe}(\mathsf{map}(X_1),\ldots,\mathsf{map}(X_k)) \tag{3}$$

$$\mathsf{map}(\mathsf{comp}(X_1,\ldots,X_k) \equiv \mathsf{comp}(\mathsf{map}(X_1),\ldots,\mathsf{map}(X_k)) \tag{4}$$

$$\mathsf{map}(\mathsf{pipe}(X_1,\ldots,X_k)) \equiv \mathsf{map}(\mathsf{comp}(X_1,\ldots,X_k)) \tag{5}$$

$$\mathsf{pipe}(\mathsf{map}(X_1),\ldots,\mathsf{map}(X_k)) \equiv \mathsf{comp}(\mathsf{map}(X_1),\ldots,\mathsf{map}(X_k)) \tag{6}$$

Rules 3 and 4 concern map distribution with respect to pipe and comp. Rules 5 and 6 insert and remove pipeline parallelism within or immediately outside maps. It is easy to convince ourselves that the rules above actually hold. In particular, rules 5 and 6 hold because pipes and comps compute the same function, with different amounts of parallelism exploited as stated in Section 2. Rules 3 and 4 just state map distributivity with respect to both pipes and comps.

By using the rewriting rules stated above, we can get the following diagram, concerning the transformations of a simple skeleton program:

$$\begin{array}{ccc}
\mathsf{pipe}(\mathsf{map}(\mathsf{seq}(A)),\mathsf{map}(\mathsf{seq}(B))) & \longleftrightarrow & \mathsf{map}(\mathsf{pipe}(\mathsf{seq}(A),\mathsf{seq}(B))))) \\
\updownarrow & & \updownarrow \\
\mathsf{comp}(\mathsf{map}(\mathsf{seq}(A)),\mathsf{map}(\mathsf{seq}(B))) & \longleftrightarrow & \mathsf{map}(\mathsf{comp}(\mathsf{seq}(A),\mathsf{seq}(B)))
\end{array}$$

Both skeleton nestings in the first row exploit pipeline parallelism. In the first one this is actually the main (outermost skeleton) parallelism form taken into account. In presence of a small number of processing elements to execute the program, such parallelism exploitation pattern will be the only one actually exploited. The skeleton nestings in the lower row, instead, do not exploit pipeline parallelism at all. All the pipelines have been transformed into sequential composition of (possibly parallel) functions via the comp skeleton. In the lower, leftmost skeleton composition, the processing elements computing the first map are re-used to compute the second map. In the lower, rightmost skeleton composition the processing elements used to implement the map workers, first compute the sequential code of A and then the sequential code of B.

Now the interesting point is: how can we decide whether or not it is the case of applying any of the matching rewriting rules to a given skeleton program? How can we decide how a schema such as the one just discussed has to be traversed?

Assume we have a program with a skeleton nesting matching the $\mathsf{map}(\mathsf{pipe}(X_1, \ldots, X_k))$ structure. It's worthwhile, in terms of performance, to apply the rule ($5^{\rightarrow}$) and remove the pipeline parallelism or this transformation does not make any sense? Can we state a set of conditions (concerning either the parameters of the skeletons in the rule or the skeleton context where they appear) that make the transformation worthwhile? In other words: being $\mathcal{P}$ some kind of measure of the performance of a skeleton implementation, given any rewriting rule $Lhs \equiv Rhs$, can we give a set of conditions C on the structure of Lhs, Rhs and on the features of the target architecture in such a way that the formula $C \supset (\mathcal{P}(Lhs) > \mathcal{P}(Rhs))$ holds?

In order to be able to answer such kind of questions, we will discuss in the next Section a performance model framework extending the classical model used within the P3L framework. Then, in Section 5, we discuss how the model can be used to evaluate "in insulation" the performance effect of applying the rewriting rules presented above, especially focusing on the transformations between stream parallelism and data parallelism. The evaluations presented in that Section can be suitably used when programs with a limited skeleton nesting have to be transformed. Finally, in Section 6 we will present some arguments aimed at understanding how the performance model of Section 4 can be used to optimise skeleton program performance by properly using the rewriting rules presented in this Section.

4 Performance models

A performance model is a formal way of computing some performance measures out of a set of basic cost parameters. The performance model we want must be able to model performance measures such as the *completion time* (the time elapsed from the moment the first input is read and the moment the last output is delivered), the *service time* (the time elapsed between the moment an output is delivered and the moment the next output is delivered) and the *efficiency* (the speedup per processing element, basically). Furthermore, they must be "precise enough" to allow us to accurately study things such as the communication grain effect or the impact of pipeline parallelism within data parallel computations.

In this work, we consider an extension of the performance models traditionally used in the P3L research group [7, 12]. These models have been shown powerful enough to model performance measures such as the service time of templates. The extension consists in having a larger number of basic cost parameters and using the completion time as the main performance measure for optimizations, as opposed to the service time alone.

Therefore we adopt as cost parameters variables like the time spent in broadcasting b bytes to a set of k processes, the average amount of time spent in computing a portion of (user) sequential code, or the amount of data needed by each virtual processor in a map skeleton, as well as things such as the template *capacity*, i.e. the amount of tasks needed to make an implementation template compute according to its steady state behaviour.

By using this kind of basic parameters we model the completion time of P3L templates (relative to a target machine made out of the complete interconnection of a large number of processors) by formulas having a well defined structure, namely:

$$T_C = (L_S + M)T_S$$

where T_C stands for the completion time, L_S is the overall number of tasks the template must process, M is the *capacity* of the template in terms of tasks buffered at the steady state and T_S stands for the service time of the nested template. The capacity parameter is the real new parameter with respect to the classical P3L performance models. By properly using the capacity we are able to devise proper performance models for the completion time in terms of the service time. The capacity and completion time of a P3L program are derived by inductively composing formulas relative to all the templates used to implement the skeletons in the program.

We developed a full set of performance models using this parameters [11]. These models will not be described in detail in this work. We just want to demonstrate the usage we can make of such models while analysing the performance effects of skeleton program rewriting. Therefore, in Section 5 we will discuss how the models can be used to analyse the performance behaviour of two equivalent skeleton programs obtained by applying some of the rules of Section 3.

The formulas derived by using our performance models looks like to be complex (see Section 5). However, we succeeded in deriving an analysis for all the templates taken into account here. This kind of template behaviour analysis is exactly what we need in order to be able to study the rewriting rules of Section 3. We could have used different cost models, such as PRAM-like or BSP models [13]. Those models abstract the machine behaviour by using a much smaller parameter set and therefore they lead to more "concise" performance formulas. However, such models do not seem to be able to distinguish all the parameters we need in order to perform precise analysis of the effects of rewriting rules on performance. Actually, within our research group some researchers are investigating whether some slight variation of BSP-like models can be used to perform more efficiently the kind of cost analysis we describe in this paper.

5 Evaluation of rewriting rules via the performance models

In this Section we will show how the performance model framework outlined in Section 4 can be used to evaluate the effectiveness of a rewriting rule such as the ones discussed in Section 3.

We will discuss two cases: in Section 5.1 we will discuss the performance effects of the map promotion rule (3) while in Section 5.2 we will discuss the effects of transforming `pipeline` stream parallelism into `comp` data parallelism via the rule (5).

5.1 Map promotion rule (rules ($3^{\rightarrow}$) and ($3^{\leftarrow}$))

The templates corresponding to the two sides of the rule are shown in Figure 2. It is clear that the `pipe` of `map` template composition presents a higher number of "synchronization" points due to the `E` and `C` processes. Such syncronisation points seem to make the `pipe` of `map` template inherently slower than the `map` of `pipe`. However, due to the fact that pipeline parallelism is exploited, using different processing elements to compute the different X_i in the `pipe` of `map` template composition, we can expect that the time loss due to the emitter/collector processes is overwhelmed by the time gain obtained from parallel computations of the stages X_i. Furthermore, each one of the `map`s of the `pipe` of `map` side can be optimized (e.g. by devising a different number of worker processes), thus

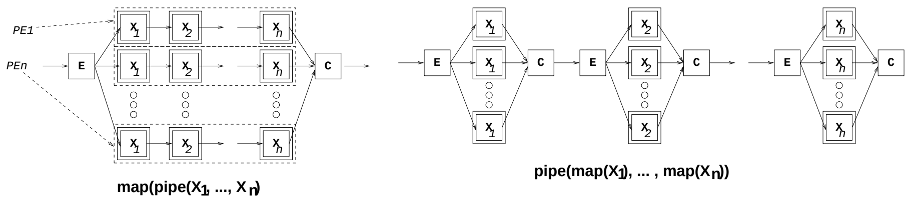

Fig. 2. Template implementation of left and right hand side of rule (3)

achieving a better balancing of the computation. And this better balancing will eventually lead to a better overall performance of the template.

By using our performance models we eventually come out with the completion time formulas for the pipe of map and map of pipe template composition, that is:

$$T_c^{mp} = (L_s + M^{mp})T_s^{mp} \qquad vs \qquad T_c^{pm} = (L_s + M^{pm})T_s^{pm}$$

We denote with ϕ^{mp} ($\phi \in \{T_C, T_S, M\}$) the value of function ϕ for the map(pipe$(\ldots)$) and with ϕ^{pm} the value of function ϕ for the pipe(map$(\ldots),\ldots)$. In the general case, ϕ^X will refer to the value of function ϕ in the skeleton X.

By instantiating all the formula parameters and looking at the number of used PEs that minimizes the completion time, we see that the service time is almost the same in the two cases. Therefore, to look for the better completion time we must look at the behavior of the capacity of the two template compositions. If we take into account the amount of resources used (the amount of processing nodes needed to implement the template compositions, actually), and the number of tasks that have to be present onto the input stream in order to achieve the steady state of the template, the pipe of map and the map of pipe template are very different.

In the two cases, the number of tasks needed to reach the steady state turns out to be:

$$M^{mp} = 2nk_e + \sum_{i=1}^{n} \left(\frac{M^{X_i}}{k_i^{X_i}} \right) \qquad vs \qquad 2k_e + \frac{\sum_{i=1}^{n} M^{X_i}}{k_i} = M^{pm} \tag{7}$$

where: $k_i^{X_j}$ represents the number of virtual processors scheduled for execution on a PE of template X_j, k_e represents communication grain and n is the number of pipeline stages.

In pipe of map template, the maximum parallelism (the minimum number of virtual processors per PE) is achieved in the pipeline stage implementing the skeleton with the highest service time. Similarly, the unique map template of the map of pipe template will be dimensioned with the same parallelism degree, as its service time will be the maximum of the service times of the inner stages.

Furthermore, we must point out that:

– the terms $2k_e$ and $2nk_e$ are due to the E and C processes of the maps. The map of pipe term is obviously always better than the corresponding pipe of map term. The skeleton nesting map(pipe(...)) is always more efficient of the pipe(map(...), ..., map(...)) one, according to this viewpoint. This is because data structures are decomposed and distributed just once in the map of pipe template (possibly using expensive communications), whereas in the pipe of map template the decomposition/distribution is performed n times.
– the terms $\left(\sum_{i=1}^{n} M^{X_i}\right)/k_i$ and $\sum_{i=1}^{n}\left(M^{X_i}/k_i^{X_i}\right)$ represent the capacities introduced by the replication of pipe in the stage of the map worker (map of pipe template) and by the replication of the map within each pipeline stage (pipe of map template), respectively. The map of pipe term is always worst than the pipe of map one as for every $i \in (1, n)$ it holds that $k_i \leq\ k_i^{X_i}$. In fact, the skeleton nesting map(pipe(...)) is less correct with respect to the parallelism exploited, because all the pipeline stages are executed with the parallelism degree of the stage having the higher service time, thus leading to a capacity larger than the one actually needed to exploit all of the stage parallelism.

Although some "qualitative" reasoning can actually be performed looking at the performance model formulas, in order to fully assert the behaviour of the two template compositions we need to exactly evaluate the two formulas, providing all the parameters and looking at the numerical result. However, this is not too much useful. Many parameters can be supplied just after having completely specified the skeleton parameters. Instead we are interested in general guidelines stating whether, under a precise set of conditions, a rule can be conveniently applied or not. Such guidelines can be obtained restricting the variability of parameters, or, in other words, simplifying the formulas in such a way that well defined cases are modeled instead of the general ones.

In the following two subsections, we will discuss in particular the rewriting rules ($3^{\rightarrow}$) and ($3^{\leftarrow}$) in case two different hypothesis hold: the X_i skeletons are plain sequential skeletons or they are arbitrary skeleton compositions such that all of them take a comparable service (completion) time.

Sequential X_i computations Suppose the different X_i skeletons are sequential skeletons. This is a significant case, as independent data parallel computations over large data structures usually allow high degrees of parallelism to be exploited, making useless any further parallelism exploitation within the workers computing the X_i.

The skeleton nesting pipe(map(...), ..., map(...)) looks like to be asintotically better, due to the fact that every stage of the pipeline can be independently optimized. The price to pay is the high number of "synchronising" processes inserted in the template composition. Such synchronising processes (the E and C processes in Figure 2, right), may cause a minimum performance loss in case the subtrees computing the different X_i are very complex. However, in case all the X_i are computed by sequential code, the performance penalty due to the synchronization introduced by the processes turns out to be very high. In this case, using our models we derived the following formulas:

$$M^{mp} = (n+1)k_e + \sum_{i=1}^{n} \frac{M^{X_i}}{k_i} \qquad vs \qquad 2nk_e + \sum_{i=1}^{n} \frac{M^{X_i}}{k_i^{X_i}} = M^{pm}$$

As the capacity of a sequential template is constant and equal to 1, and the number of virtual processors allocated over a single node must always be less than the overall number of virtual processors ($1 \leq\ k_i^{X_i} \leq\ d$) we eventually get:

$$k_e \geq\ n/(n-1) \quad \Rightarrow \quad M^{mp} \leq\ M^{pm}$$

and therefore we can conclude that the map of pipe template composition always performs better that the other one.

Balanced X_i computations Now take into account the case of having non-sequential X_i skeletons, but assume that the different X_i computations all take a similar ("balanced") service time. This is the case we can achieve by using farm-introduction rule over the "heavy" X_i skeletons, for instance.

Under this assumptions, the number of virtual processors allocated to the PEs running the map workers in the map of pipe skeleton tree will be equal to the one used in the maps of the pipe of map template composition. The capacity and, as a consequence, the startup times of the nested template will be comparable, therefore the pipe of map template composition will pay a high overhead due to E and C processes. Transposing this reasoning into formulas we eventually get:

$$M^{mp} = 2k_e + \frac{2\sum_{i=1}^{n} M^{X_i}}{\Psi} \quad \leq \quad 2nk_e + \frac{2\sum_{i=1}^{n} M^{X_i}}{\Psi} = M^{pm}$$

$$M^{mp} = 2k_e \quad \leq \quad 2nk_e = M^{pm}$$

where Ψ essentially represents the number of virtual processor in every map, i.e. in this case also it is always convenient use the map of pipe template composition with respect to the pipe of map one.

5.2 Transforming pipes into comps within a map (rule ($5^{\rightarrow}$))

We consider now the rule ($5^{\rightarrow}$) of Section 3. This rule states that a pipeline skeleton within a map skeleton can be substituted by a comp skeleton. In other words, the rule states that pipeline parallelism within a map can be neglected and plain sequential computations can be performed instead. We restrict the analysis to the cases the different X_i are either sequential modules or data parallel modules.

By looking at the performance models of the completion time of left and right hand side of the rule, we observe that in case the two sides are dimensioned with respect to the minimal completion time, they are basically equivalent. The parallelism we loose in removing the pipe skeleton is restored by using a larger amount of PEs in the implementation of the maps in the pipe of map template. This holds until we reach the limit of the amount of PEs that can be included in a map template (which is equal to the number of the virtual processors defined by the map skeleton).

Due to the higher service time of the comp, such limit can be reached in the map of pipe and map of comp with different values of the parameters. By looking at the models, we know that reaching the map PE limit in the left hand side implies that the limit (using the same parameters) should have been reached even in the right hand side. However the viceversa is not true. As a result, we can conclude that:

- in case the communication/computation ratio represents the bottleneck, the two sides of the rule are roughly equivalent,
- in the other cases (i.e. when a virtual processor computes a lot with respect to the time it spends in communications) the map of pipe presents a better completion time than the right hand side.

6 Using the "evaluated" rewriting rules

The kind of reasoning developed in Section 5 can be usefully applied when programs are poorly structured. In case programs contain simple skeleton nestings, the programmer may look at the results of this kind of analysis and decide, as an example, whether or not the program he originally conceived as a pipe of maps may usefully be turned into a map of pipes.

When larger programs are considered (i.e. programs with deeper skeleton nestings), the "local" optimisations we evaluated in Section 5 may lead to mistakes. As usual, local optimisations may drive the programmer towards a local minimum rather than to the global one. We are currently looking for algorithms (heuristic based search algorithms, basically) able to find some kind of "optimal" transformation of a skeleton program among all the valid transformations, but we have no concrete results, yet. Therefore, we do not pretend to present a general algorithm. Instead, here we want to point out how an expert skeleton programmer can take advantage of the methodology aimed at evaluating the rewriting rules discussed in this paper, by using a simple example.

Suppose we have a program originally structured as pipe(farm(map(T_1)), map(T_2)) and suppose we know all the parameters needed to instantiate and evaluate the performance formulas relative to the program, derived by using the models of Section 4. By working out the skeleton structure of the program with the rewriting rules of Section 3 we eventually get a graph such as the one of Figure 3. In this graph some skeleton programs semantically equivalent to the original one are shown. Vertically or horizontally adjacent skeleton programs represent programs equivalent modulo the application of a single rewrite rule, but for those separated by a $\rightarrow^*$ arrow, requiring more than a single rewriting step. Each skeleton program has associated the expected completion time and the efficiency, derived by using the models discussed in Section 4. In the Figure

$$
\begin{array}{ccccc}
\begin{array}{c}(\mathbf{T_c = 1054}, \mathcal{E} = \mathbf{35}\%)\\ \mathsf{pipe}(\mathsf{map}(\mathsf{farm}(T_1)), \mathsf{map}(T_2))\end{array} & \rightarrow_{\mathcal{E}} & \begin{array}{c}(T_c = 1066, \mathcal{E} = 46\%)\\ \mathsf{pipe}(\mathsf{map}(T_1), \mathsf{map}(T_2))\end{array} & \rightarrow_{\mathcal{E}} & \begin{array}{c}(T_c = 931, \mathcal{E} = 49\%)\\ \mathsf{map}(\mathsf{pipe}(T_1, T_2))\end{array} \\
\downarrow_{T_c} & & & & \downarrow_{\mathcal{E}} \\
\begin{array}{c}(T_c = 1007, \mathcal{E} = 31\%)\\ \mathsf{pipe}(\mathsf{map}(\mathsf{farm}(T_1)), \mathsf{map}(\mathsf{farm}(T_2)))\end{array} & & \begin{array}{c}(T_c = 1027, \mathcal{E} = 38\%)\\ \mathsf{pipe}(\mathsf{map}(T_1), \mathsf{map}(\mathsf{farm}(T_2)))\end{array} & (\rightarrow^*) & \begin{array}{c}(\mathbf{T_c = 875}, \mathcal{E} = \mathbf{66}\%)\\ \mathsf{map}(\mathsf{comp}(T_1, T_2))\end{array} \\
\downarrow_{T_c} & & & & \\
\begin{array}{c}(T_c = 904, \mathcal{E} = 28\%)\\ \mathsf{map}(\mathsf{pipe}(\mathsf{farm}(T_1), \mathsf{farm}(T_2)))\end{array} & & \begin{array}{c}(T_c = 1036, \mathcal{E} = 12\%)\\ \mathsf{map}(\mathsf{pipe}(T_1, \mathsf{farm}(T_2)))\end{array} & (\rightarrow^*) & \begin{array}{c}(T_c = 1052, \mathcal{E} = 46\%)\\ \mathsf{farm}(\mathsf{map}(\mathsf{comp}(T_1, T_2)))\end{array}
\end{array}
$$

Fig. 3. Transformation of the original example program by using the rewriting rules (completion times are in $m$secs)

we also show two paths. The path labeled with $\rightarrow_{\mathcal{E}}$ corresponds to the path we follow if at any moment we choose to apply the more convenient transformation rule (with respect to efficiency) matching the inner skeletons. The path labeled with $\rightarrow_{T_c}$, instead, corresponds to the path we follow if we apply a similar algorithm looking for the better completion time. In other words we apply in the two cases a greedy algorithm to the skeleton tree representing the program.

We see that if we try to optimize the completion time we eventually come to the lower left skeleton program. This program does not represent a "good" transformation of the original one, as the efficiency is really poor. By looking at the overall picture, we easily see the program transformation we reached is a local minimum. We could move one step right from the leftmost top skeleton composition by applying farm elimination rule, but this move actually increases the target function value, i.e. the completion time. If we make this move, we can subsequently move one step right and one down right and reach the program transformation with the minimum completion time. Instead, if we follow the algorithm trying to optimize efficiency, starting at the original program node we eventually come to the skeleton program ($\mathsf{map}(\mathsf{comp}(T_1, T_2))$) which is the one with the better completion time and efficiency. The communication/computation ratio happens to be below 0.3, both for the first and for the second map, in this case, and the input stream we take into account is quite "short". These are the reasons why we almost always get an improvement when eliminating pipeline parallelism from the program. If the stream is longer, we obtain a different graph.

Although this example clearly demonstrates the failure of greedy search algorithms due to local minimums, it shows the kind of reasoning a programmer may perform once he has available all the parameters needed to instantiate the performance model formulas, *before* he actually has to write a single line of code.

7 Conclusions

There is a problem, when using program transformations in a skeleton framework: functionally equivalent programs exploit different kind of parallelism. In order to understand which one of a set of equivalent programs is the best one, we need to establish evaluations of rewriting rules (in terms of the performance achieved), i.e. we need to set up a framework telling us whether or not a rule can be conveniently applied (left-to-right or right-to-left).

Here we outlined a performance model that allows such evaluations to be performed. In particular we showed how the performance model can be used to evaluate program transformations affecting the basic kind of parallelism exploited, i.e. those rules changing stream parallelism to data parallelism and viceversa. We presented some of these rules and discussed how they can be "evaluated", i.e. how a generic, bi-directional rewriting rule can be transformed into a uni-directional rule with preconditions. In case the preconditions are satisfied, the bi-directional rule is to be used just in one direction, if we want to achieve a better program performance. Eventually, we showed how the framework discussed in the first part of the paper can be used to reason onto "the real program at hand".

The ideas behind our transformations can be found in many other skeleton related works. Most of these works, however, just take into account data parallel skeletons and, as a consequence, they do not address any kind of transformation between stream and data parallelism [14, 15, 16]. Furthermore, these works often use "asymptotic" cost calculus [17, 15]. Our cost calculus, although more complex, looks like to be more suitable to model real machines. The approach we take in performing transformations is closer to Gorlatch's work [18]. The transformations he uses to remove unnecessary synchronisation points between consecutive stages of a computation are similar to the transformation we perform with the rewrite rules of Section 3.

The performance models discussed in Section 4 are original, as they represent a concrete evolution of the models used until now within the P3L project, that only model the service time of template and do not take into account the completion time, built out of the service time and capacity, as a performance measure to optimize. These models have still to be extensively validated. Our experience and some restricted experimental results make the models look like to be correct. Currently we are trying to achieve a full validation of the models via the simulation of the templates onto a real parallel architecture.

The main result is that we have shown a possible way to set up a framework able to discriminate stream parallel and data parallel skeleton compositions with respect to performance.

References

[1] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[2] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, July 1992.

[3] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Third Parallel Computing Workshop (PCW'94)*. Fujitsu Laboratories Ltd., November 1994.

[4] D. B. Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2), April 1991.

[5] D. R. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T von Eicken. LogP: towards a realistic model of parallel computation. In *ACM/SIGPLAN PoPP*, 1993.

[6] S. Pelagatti. A methodology for the development and the support of massively parallel programs. Technical Report TD-11/93, Dept. of Computer Science – Pisa, 1993. PhD Thesis.

[7] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[8] B. Bacci, M. Danelutto, and S. Pelagatti. Automatic Balancing of Stream-Parallel Computations in P3L. Technical Report TR-35/93, Department of Computer Science, University of Pisa (Italy), 1993. Available on ftp anonymous at the site `ftp.di.unipi.it`.

[9] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

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

[11] M. Coppola and M. Aldinucci. Optimisation techniques for structured parallel programs. Graduation thesis (in italian), Department of Computer Science, University of Pisa, 1997.

[12] B. Bacci, M. Danelutto, S. Pelagatti, S. Orlando, and M. Vanneschi. Unbalanced Computations onto a Transputer Grid. In *Proceedings of The 1994 Transputer Research and Application Conference*, pages 268–282. IOS Press, October 1994. Athens, Georgia, USA.

[13] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, August 1990.

[14] Eric Violard, Stephane Genaud, and Guy-Ren Perrin. Refinement of data parallel programs in pei. In Richard Bird and Lambert Meertens, editors, *IFIP TC2 Working Conference on Algorithmic Language and Calculi*. Chapman & Hall, February 1997.

[15] H. Deldarie, J. R. Davy, and P. M. Dew. The performance of parallel algorithmic skeletons. Technical Report 95/6, University of Leeds, School of Computer Studies, 1995.

[16] C.B. Jay, D.G. Clarke, and J.J. Edwards. Exploiting shape in parallel programming. In *1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing: Proceedings*, pages 295–302. IEEE, 1996.

[17] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[18] S. Gorlatch. Stages and transformations in parallel programming. In *Abstract Machine Models*, pages 147–161. IOS Press, 1996.