EVALUATING COMPUTATIONAL COSTS WHILE HANDLING DATA AND CONTROL PARALLELISM

SONIA CAMPA
Dept. Computer Science, University of Pisa, Largo B.Pontecorvo 3
Pisa, 56123, Italy

ABSTRACT

The aim of this work is to introduce a computational costs system associated to a semantic framework for orthogonal data and control parallelism handling. In such a framework a parallel application is described by a semantic expression involving in an orthogonal manner both data access and control parallelism abstractions. The evaluation of such an expression is driven by a set of rewriting rules each of which is combined with a computational cost. We present how to proceed in the evaluation of the final cost of the application as well as how such information together with the semantic framework capabilities can be exploited to increase the overall performance.

1. Introduction

In the latest years, the literature has widely recognized all the advantages provided by the structured parallel programming paradigm [1] in which an application is given as a composition of (a set of) recurrent patterns whose control behavior is well known and predictable, while the functional behavior depends on the functional parameters provided by the user who is instantiating the patterns. Stating in advance the control behavior of an application allows, in principle, to statically make some kind of prediction on its computational costs and thus, to intervene on the application graph in order to increase the overall performance (for example, by augmenting the parallelism degree). On the other hand, a parallel application have also to deal with data access concerns that can heavily influence both the programming phase and the final computational costs. Unfortunately, the most works regarding structured parallel programming environments, define a two-tier architecture model in which data accesses have to be heavily detailed on top of control abstractions [2,3] or vice versa [4,5,6,7]. As a consequence, the model of parallelism is a mixture of strictly coupled data and control concerns difficult to formalize and, hence, to statically and/or dynamically analyze for optimization purpose.

In [8] we have sketched a programming model in which parallel applications are built by keeping orthogonal data and task parallel concerns through a set of abstract mechanisms. In [9] we have given a preliminary formalization of the model, pointing out that the evaluation of a parallel application is described by a sequence of transformations (i.e. inference rules) on the application graph involving abstract mechanisms for both expressing control concerns and data access concerns. The contribution given by this work is a more clear specification of the semantics and the introduction of a computational costs system provided by the semantics. In fact, the former formalizes both mechanisms for describing control concerns and data access concerns, thus reaching three goals:

- giving a formal description of the abstract mechanisms that handle data and task parallel concerns leads to a clear, unambiguous description of the programming model, regardless any further implementation choice
- the semantics can assign some kind of evaluation costs about the use of our abstract mechanisms use in order to statically estimate the computational cost of the whole application.
- the semantics is enriched with a set of rewriting rules about graph structures that assess if two graphs are functionally equivalent and each graph has associated an estimation of its computational cost. Rewriting rules and static estimations can drive static and dynamic improvements of the overall performance. For example, it can be proved that a certain control structure is equivalent and more efficient than the one declared by the user, thus driving a transparent rewriting process of the graph application at running or compile time.

This work is structured as follows: Section gives an overall introduction to the programming model we are assessing. Section presents the basic of both our semantics and our costs system, in particular which operators are involved as well as how they can be used to describe and computationally evaluate parallel applications. Section shows a simple case study as well as how our cost system can lead to the estimation of the final computational costs. Section concludes the presentation by giving an overview of related and future work.

2. The programming model

Our programming model is fully described by the tuple $\mathcal{M} =< \mathcal{A}, \mathcal{V}, \mathcal{I}, \mathcal{C} >$ where $\mathcal{A}$ stands for a set of *abstract data types*; $\mathcal{V}$ stands for the set of *view*s on abstract data types; $\mathcal{I}$ stands for a set of *iterators* for accessing views; $\mathcal{C}$ stands for the set of *collectives* (or *control primitives*) which allow to describe structured functional graphs (i.e. graphs built composing the available collectives).

The first three mechanisms fully describe the data parallel concerns of the application. An instance of a given abstract data type is a representation of the raw input data abstracting from their actual implementation and/or distribution. Different kind of typed views can be declared on top of such instance, thus providing a logical organization of the raw input data and a set of operators to manipulate them. Since the view is independent from the actual implementation of data, such separation gives many hooks to optimize the implementation of both the view's

operators and the data itself.
Each view provides a set of typed iterators on its items. An iterator is an object exposing a set of operators to get items from the view, coherently with the view type and with a given pattern of access. Moreover, each type of iterator can be specialized in order to get the items in a whole or sequentially. For example, an array view provides iterators to get singletons or block of items, while a tree view provides iterators to get subtrees, children or siblings of a given node and so on. Such items can be returned by the iterator one-by-one or as a whole, depending on the specialized behavior of the iterator.

The task parallel behavior of the application is described by using the set of primitives $\mathcal{C}$, a collection of composable patterns of control plus sequential functions types. The basic idea is that the user application can be formalized as a graph in which the nodes are sequential or parallel modules and the arcs are functional (i.e. data) dependencies among modules. Such an application graph is constructed by selecting and composing the patterns provided by the set of primitives $\mathcal{C}$. `Pipe` (pipelines of stages), `Apply-to-all` (application of a function to all the elements of an input data set in parallel), `Comp` (sequential composition), `Reduce` (function reduction) etc. populate $\mathcal{C}$.
For example, let us suppose we have an application that applies the function $S_2(S_1(px))$ to each pixel px of the images belonging to an input stream of images. Such application can be written as a pipeline of two stages: the first stage *applies* $S_1(px)$ *to all* the pixels of the current image; the second stage *applies* S_2 *to all* the pixels updated by the first one. Hence, the global application results in a `Pipe` of two stages, each of which is an `Apply-to-all` computing a given sequential function.

Summarizing, our programming model allows to exploit data parallelism by selecting appropriate abstract data types, views and iterators and without keeping in mind how to evaluate the functions that will use them. On the other hand, the application can be completely described by composing its computational graph regardless details about accesses and/or distribution of input data.
Nevertheless, since each module in the graph accesses only the piece of data that flows from its parent, its input can be easily provided by an iterator handling such portion, thus coupling in a very high level manner control and access behavior. In fact, in our programming model the harmonization between data and task parallelism is given by coupling each selected primitive with one or more iterators. The internal behavior of each primitive depends on the specific pair control pattern-iterator by which it is composed. For example, let us consider an application that can be evaluated applying in parallel the same function to all the items belonging to an array view. As mentioned before, such simple pattern is represented in $\mathcal{C}$ by the `Apply-to-all` primitive. However, more pragmatically, the application can be written instantiating a set of processing elements each of which applies the function to an independent task of the input data set, thus exploiting a plain task parallel behavior. Otherwise, the same application can be written as a plain data parallel computation, thus applying the function to all the items of the data set in parallel. The former specialized pattern is called `farm` in the skeleton community while the latter is called `map`. In our model, a `farm` computation can be described by

combining the `Apply-to-all` primitive with a sequential iterator (thus, applying a function to all the elements accessed one-by-one). Instead, a `map` computation can be obtained by coupling the same primitive with a parallel iterator: in this case all the elements are accessed as a whole and the function is applied in parallel on each of them.

3. Semantics

The formal semantics of our programming model is given by populating the tuple $< \mathcal{A}, \mathcal{V}, \mathcal{I}, \mathcal{C} >$. Each type of the tuple is represented as a set plus some operators to manipulate its elements.
Since $\mathcal{A}$ is the type representing the raw input data, it can be modeled as a set of objects of a generic type $\mathcal{O}$ provided with canonical operators for creating a set, querying its content, inserting, deleting, updating elements, and so on. Tab.1 summarize the formalization of type $\mathcal{A}$. The view abstraction is provided by the

$\mathcal{A} = \{o_1, \ldots, o_n \mid o_i : \mathcal{O}\}$	
$create : () \to \mathcal{A}$	$add : \mathcal{A} \times \mathcal{O} \to \mathcal{A}$
$size : \mathcal{A} \to \texttt{Nat}$	$\ldots$

Table 1. $\mathcal{A}$ type definitions

$\mathcal{V} = \{$`Array, Matrix, Graph, Tree, List` $\}$	
$create : \mathcal{A} \to \mathcal{V}$	$getIterator : \mathcal{V} \to \mathcal{I}$
$\ldots$	$\ldots$

Table 2. $\mathcal{V}$ type definitions

$\mathcal{I} : \mathcal{V} \to (\mathcal{V} \times \texttt{Nat})$	
$curr : \mathcal{I} \to \mathcal{O}$	$skip : \mathcal{I} \to \mathcal{I}$
$hasNext : \mathcal{I} \to Bool$	$set : \mathcal{I} \times \mathcal{O} \to \mathcal{I}$
$\ldots$	$\ldots$

Table 3. $\mathcal{I}$ type definitions

set $\mathcal{V}$ of view types that map a given $\mathcal{A}$ object to a $\mathcal{V}$ object. At the moment, the provided view types are `Array`, `Matrix`, `Graph`, `Tree` and `List`. For example, given $s = \{s_1, \ldots, s_{size(s)}\} \in \mathcal{A}$, and given $g \in \texttt{Nat}$ as a grain value divisor of $size(s)$, the `Array(s,g)` type is constructed by the mapping function

$$\mathcal{A} \times \mathcal{N} \to \texttt{Array}(\texttt{s}, \texttt{g}) = << s_1, \ldots, s_g >, \ldots, < s_{size(s)-g}, \ldots, s_{size(s)} >>$$

Each view $v \in \mathcal{V}$ is a factory for a set of iterators on its structure, thus it provides kind of `getIterator` operators (see Tab.2). Generally speaking, an iterator is a pair (v, p) where $v \subseteq \mathcal{V}$ is the (sub)view on which it is defined and $p \in \texttt{Nat}$ is a pointer to the next accessible item of its range.
Tab. 3 shows some operations to handle an iterator object. The *curr* operator returns the current item pointed by p. If the iterator has a parallel behavior, p "points" in parallel to all the items so that they are taken as a whole; otherwise p points also to the current item and the operator returns it as a singleton.

The set of collectives $\mathcal{C}$, as mentioned before, includes well known patterns of control as for example `Pipe`, `Apply-to-all`, `Comp` etc. Generally speaking, each collective can be viewed as a function

$$\mathcal{C} : \mathcal{I} \times (\mathcal{C} \cup \mathit{Function}) \rightarrow \mathcal{V}$$

that gets a sequential function (let us call `Function` its type) or a nested control primitive and a set of values (i.e. the ones given by one (or more) iterator(s)) and returns the view obtained by applying the second argument to the set of values. As explained in Section , the parallel access behavior by which such values will be accessed is encapsulated by the input iterator instance.

In our semantic model, collectives, iterators and related operators are defined in terms of semantics expressions whose evaluation is given by means of inference rules stating functional equivalences between expressions. T_1 is a general example of a rule of ours:

$$T_1 : \frac{C(E_1)}{E_1 \rightarrow_c E_2}$$

The expression E_1 is transformed by rule T_1 into the equivalent expression E_2, if some condition C on E_1 holds. Each transformation is provided with a cost c and the semantics supplies a cost function $\mathcal{E} : \mathcal{C} \rightarrow \texttt{Nat}$ that associates a costs to each primitive $\mathcal{C}$ in the following way. Let $\mathcal{C} \rightarrow_{c_1} E_1 \rightarrow_{c_2} \ldots \rightarrow_{c_k} v$ be the chain of transformations for evaluating $\mathcal{C}$ up to the final result v. Then,

$$\mathcal{E}(\mathcal{C}) = \sum_{i=1}^{k} c_i \tag{1}$$

that is, the computational cost for evaluating $\mathcal{C}$ is the sum of the costs of all the partial transformations applied to transform $\mathcal{C}$ into v.

Let us give some example of how this relation works and on which operators. As mentioned before, the iterator is represented as a pair $(\overline{x}, p)$ where $\overline{x} =< x_1, \ldots, x_n >$. The *curr* operator is in charge of returning (thus is functionally equivalent to) the current element of its range. If the iterator *it* is an instance of *SeqIterator*, then *curr* will only return the p-th element of the range; if the *it* is of type *ParIterator*, then the whole range is returned. The *curr* operator is evaluated by the following rules:

$$\frac{it = (< x_1, \ldots, x_n >, p) : \mathit{SeqIterator}}{curr(it) \rightarrow_{c_{curr}} x_p} \qquad \frac{it = (x, p) : \mathit{ParIterator}}{curr(it) \rightarrow_{c_{curr}} x} \tag{2}$$

In the formulae given above c_{curr} represent the cost for evaluating the *curr* operator. Thus, quantitatively speaking, such cost includes costs for retrieving and accessing data.

Another operator that will often occur in the following is `skip`. Its role is simply to skip the pointer to the next position in the iterator state (or returns $\perp$ if any), without returning values.

$$\frac{it = (x, p) \wedge it' = (x, p+1)}{skip(it) \rightarrow_{c_{skip}} it'} \qquad \frac{it = (x, size(x))}{skip(it) \rightarrow_{c_{skip}} \perp} \tag{3}$$

In this case c_{skip} represents just the cost for updating the internal state of *it*.

The operator that describes the parallel evaluation of two functions f and g is given by $f \mid\mid g$ (`par` operator). For example, two of the rules leading the transformation of such an operator are the ones assessing the evaluation order:

$$a.\frac{g(x) \rightarrow_{c_1} y' \wedge f(x) \rightarrow_{c_2} x' \wedge c = \max\{c_1, c_2\}}{h(g(x))||h'(f(x)) \rightarrow_c h(y')||h'(x')} \qquad b.\frac{g(y) \rightarrow_{c_1} y'}{x'||g(y) \rightarrow_{c_1} < x, y' >} \tag{4}$$

The evaluation of `par` is strict i.e. before evaluating it, all its arguments have to be completely evaluated (rule 4-a). Moreover, the evaluation of `par` produces a new view by appending all the partial results in a new data set (rule 4-b).

The semantics includes the operator `chain` ($\oplus$) that, given a set of functions $f_1, \ldots, f_m$ represented by the sequential iterator $it_f = (< f_1, \ldots, f_m >, p)$ and an object $x \in \mathcal{O}$, evaluates $f_m(..(f_1(x))..)$. Such operator is transformed as follows:

$$\frac{hasNext(it_f) \rightarrow_{c_h} true}{it_f \oplus x \rightarrow_{c_\oplus} skip(it_f) \oplus curr(it_f)(x)} \qquad \frac{hasNext(it_f) \rightarrow_{c_h} false}{it_f^{(p)} \oplus x \rightarrow_{c_\oplus} x} \tag{5}$$

The `chain` operator is called recursively on the updated state of it_f and on the last evaluated object, till there are functions in it_f to apply.

The composition of `par` and `chain` allows to define the `Pipe` collective, e.g. a pattern of control that applies the composition of the functions represented by the iterator it_f, to the stream of input values represented by the *sequential* iterator it_d.

$$\frac{hasNext(it_d) \rightarrow_{c_h} true \wedge c_h = c_{pipe}}{Pipe(it_f, it_d) \rightarrow_{c_{pipe}} it_f \oplus curr(it_d) \mid\mid Pipe(it_f, skip(it_d))} \tag{6}$$

$$\frac{hasNext(it_d) \rightarrow_{c_h} false \wedge c_h = c_{pipe}}{Pipe(it_f, it_d) \rightarrow_{c_{pipe}} <>} \tag{7}$$

As it can be seen, for each transformation call of `Pipe`, the evaluation proceeds in two parallel steps [*]: the `chain` operator is applied to the functions provided by it_f and the value currently pointed by it_d through the *curr* operator. In the meanwhile, a recursive call of the operator `Pipe` is evaluated together with the updated state of it_d (rule 6). If there are no more elements to evaluate pointed by it_d, then an empty view is returned (rule 7).

4. Computational costs analysis

Just to show the effectiveness of our rules, we will give a simple evaluation example. Let $it_f = (< f_1, f_2 >, 1)$, $it_d = (< x_1, x_2, x_3 >, 1)$ and $it^{(p)}$ be the notation for iterators stressing that the current state of it points to the p-th element. Tab.4 shows the evaluation process of $Pipe(it_f, it_d)$ labelling each transformation with its cost load.

The example demonstrates that evaluating a collective through our rules, not only drives through a formal description of the transformational process from one expression to a functionally equivalent one; it also allows to model how the computation evolves in time and to define the final computational cost. In fact, in a pipeline

[*]Since the evaluation of $||$ is *strict*, all its arguments have to be evaluated, first

1	$Pipe(it_f, it_d)$	$\rightarrow_{cpipe}$	$it_f^{(1)} \oplus curr^{(1)}(it_d) \Vert Pipe(it_f, skip^{(2)}(it_d))$
2		$\rightarrow_{max\{c_{curr}, c_{skip}\}}$	$it_f^{(1)} \oplus x_1 \Vert Pipe(it_f, it_d^{(2)})$
3		$\rightarrow_{max\{c_{\oplus}, c_{pipe}\}}$	$skip^{(2)}(it_f) \oplus (curr^{(1)} it_f)(x_1) \Vert (it_c \oplus curr^{(2)}(it_d) \Vert$
			$\Vert Pipe(it_f, skip^{(3)}(it_d)))$
4		$\rightarrow_{max\{c_{curr}, c_{skip}\}}$	$it_f^{(2)} \oplus f_1(x_1) \Vert (it_c \oplus x_2) \Vert Pipe(it_f, it_d^{(3)})$
5		$\rightarrow_{\mathbf{c_{f_1}}}$	$it_f^{(2)} \oplus y_1 \Vert (it_c \oplus x_2) \Vert Pipe(it_f, it_d^{(3)})$
6		$\rightarrow_{max\{c_{\oplus}, c_{pipe}\}}$	$skip^{(3)}(it_f) \oplus curr^{(2)}(y_1) \Vert skip^{(2)}(it_f) \oplus curr^{(1)}(it_f)(x_2) \Vert$
			$\Vert it_f \oplus curr^{(3)}(it_d) \Vert Pipe(it_f, skip^{(4)}(it_d))$
7		$\rightarrow_{max\{c_{curr}, c_{skip}\}}$	$it_f^{(3)} \oplus f_2(y_1) \Vert it_f^{(2)} \oplus f_1(x_2) \Vert it_f \oplus x_3 \Vert Pipe(it_f, it_d^{(4)})$
8		$\rightarrow_{\mathbf{max\{c_{f_1}, c_{f_2}\}}}$	$it_f^{(3)} \oplus y_1' \Vert it_f^{(2)} \oplus y_2 \Vert it_f \oplus x_3 \Vert Pipe(it_f, it_d^{(4)})$
9		$\rightarrow_{max\{c_{\oplus}, c_{pipe}\}}$	$y_1' \Vert skip^{(3)}(it_f) \oplus curr^{(2)}(y_2) \Vert skip^{(2)}(it_f) \oplus curr^{(1)} it_f(x_3) \Vert []$
10		$\rightarrow_{max\{c_{curr}, c_{skip}\}}$	$y_1' \Vert it_f^{(3)} \oplus f_2(y_2) \Vert it_f^{(2)} \oplus f_1(x_3) \Vert []$
11		$\rightarrow_{\mathbf{max\{c_{f_1}, c_{f_2}\}}}$	$y_1' \Vert it_f^{(3)} \oplus y_2' \Vert it_f^{(2)} \oplus y_3 \Vert []$
12		$\rightarrow_{c_{\oplus}}$	$y_1' \Vert y_2' \Vert skip^{(3)}(it_f) \oplus curr^{(2)}(y_3)$
13		$\rightarrow_{max\{c_{curr}, c_{skip}\}}$	$y_1' \Vert y_2' \Vert it_f^{(3)} \oplus f_2(y_3)$
14		$\rightarrow_{\mathbf{cf_2}}$	$y_1' \Vert y_2' \Vert it_f^{(3)} \oplus y_3'$
15		$\rightarrow_{c_{\oplus}}$	$< y_1', y_2', y_3' >$

Table 4. Evaluation process of $Pipe(it_f, it_d)$

computation, the first step consists in evaluating $f_1(x_1)$; the second step consists in evaluating in parallel $f_2(f_1(x_1))$ and $f_1(x_2)$ and so on. Once the full regimen has been reached, all the stages evaluate in parallel. This behavior is perfectly described in our semantics. In fact, in Tab.4 we have traced in bold font, the transformational steps describing the parallelism among stages: after some preliminary evaluations (rows 1-4), the first stage is evaluated, costing c_{f_1} (row 5). Transformations 6-7 lead to the second and the third time-step of the pipeline (rows 8-11), in which f_1 and f_2 are evaluated in parallel. Moreover, the cost of these two steps is the maximum between the cost c_{f_1} of evaluating f_1 and the cost c_{f_2} of evaluating f_2.
The final computational cost of the `Pipe` operator is given by adding all the intermediate transformational costs, so that

$$\mathcal{E}(Pipe(it_f, it_d)) = 6c_h + 5c_{curr} + c_{f_1} + 2\max\{c_{f_1}, c_{f_2}\} + c_{f_2} \tag{8}$$

Thus, the generalized formula

$$\begin{aligned}\mathcal{E}(Pipe(it_f, it_d)) = \quad & (m+n+1)c_h + (m+n)c_{curr} + \\ & \textstyle\sum_{t=1}^{m-1} \max_{i\in[1,t]}\{c_{f_i}\} + \\ & \textstyle\sum_{t=m}^{n} \max_{i\in[1,m-1]}\{c_{f_i}\} + \\ & \textstyle\sum_{t=n+1}^{n+m-1} \max_{i\in[t-n+1,m]}\{c_{f_i}\}\end{aligned} \tag{9}$$

is the one evaluating the computational cost of a pipeline given an input stream of size $n \geq 1$ represented by the iterator $it_d = (< x_1, \ldots, x_n >, 1)$ and $m \geq 1$ stages, globally represented by the iterator $it_f = (< f_1, \ldots, f_m >, 1)$. The inner sums measure at each time step $i \in [1, m+n-1]$, the maximum paid cost and they add it to the cost accumulated in the preceding steps.

It has to be pointed out that in the previous example we have shown how our semantics allow to derive a computational cost as soon as a primitive has been defined. Each time such primitive recurs as part of a more complex graph or as a graph by its own, the formula statically estimated by our semantics framework can be taken into account to automatically estimate its computational cost of the (sub)graph.

Provided that the cost evaluation of each collective in $\mathcal{C}$ can be easily estimated by the semantics, we are able to quantify the computational costs of two functionally equivalent graphs and to choose the more efficient one for our execution. In fact, we can prove, for example that $\alpha(Pipe(it_f, it'_{seq}), it_{par}) \equiv Pipe(it_\alpha, it_{seq})$ (a `Map` of `Pipe` is functionally equivalent to a `Pipe` of `Map`) but also that $\mathcal{E}(\alpha(Pipe(it_f, it'_{seq}), it_{par})) < \mathcal{E}(Pipe(it_\alpha, it_{seq}))$. Hence, each time a user instantiates a pipelines of maps, the rewriting rule can be adopt to improve the performance. In other cases, as for example $\alpha(Pipe(it_f, it'_{seq}), it_{par}) \equiv \alpha((f_1; \ldots; f_m), it_p)$ (equivalence between a `Map` of `Pipe` versus a `Map` of the same stages composed sequentially), we know that the first `Map` costs more with respect to the second one: in fact, they exploit $O(m+n)$ and $O(m)$, respectively. Thus the actual convenience of instantiating one graph respect to the other depends on the input size.

5. Related work and conclusions

We have presented a semantic framework for the description of a programming model in which data access and control patterns (the ones composing the application graph) are described through independent abstract mechanisms. The model exposes a set of semantics expressions that define our mechanisms and a set transformations stating how a given expression evolves at running time. Such transformation, given as an inference rule, allows to statically define a functional equivalence between the left and the right member of the transformation. Moreover, the semantics provides a cost model assigning a cost to each transformation. As a consequence, such an inference rules system provides the basis for the definition of an evaluation function for assigning a cost to each pattern of control involved in our application, thus to the whole application itself.
Hence, our framework is able to statically (or even dynamically) evaluate the user application graph, to estimate the costs of its execution and, if it is the case, to apply smart rewriting rules to transform the user graph in a functionally equivalent graph but exposing a better performance.

The idea of formally evaluating in some way the computational cost of an application graph is not new, especially in the field of structured parallel programming. Several works has already presented in the past aiming at defining in some sense the behavior of a parallel structure [10,11] and, at possibly associating a cost value to its execution [12]. The novelty introduced with our approach is given by a semantic model able to both reproduce the evolution of an application graph execution and, in the same time, to estimate and/or rewrite such evolution in an easy and comprehensible way.
We are planning to provide our programming model with a suitable set of well known transformations that can be eventually applied to the user defined application in a transparent manner. In fact, the final goal is to have a programming framework in

which performance optimizations are automatically done on the basis of the static (and dynamic) analysis of the user application graph. On the semantic side, we are working on enriching the framework with new primitives and abstract data types as well as in describing more complex case studies evidencing the power of our approach.

References

[1] Murray Cole. *Algorithmic Skeletons: structured management of parallel computation.* Monograms. Pitman/MIT Press, Cambridge, MA, 1989.

[2] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Pattern-based parallel programming, August 2002. 2002 International Conference on Parallel Programming (ICPP-02), Vancouver, British Columbia, August 2002.

[3] Manuel Díaz, Bartolomé Rubio, Enrique Soler, and José M. Troya. Integrating task and data parallelism by means of coordination patterns. *Int. Conf. on Parallel Programming (ICPP-02)*, LNCS 2026:16, 2001.

[4] Ian Foster, David R. Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A library-based approach to task parallelism in a data-parallel language. *Journal of Parallel and Distributed Computing*, 45(2):148–158, 15 September 1997.

[5] H. Kuchen. A skeleton library. In *Proc. of Euro-Par 2002*, volume 2400 of *LNCS*, pages 620–628, 2002. Paderborn, Germany.

[6] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141, June 2002.

[7] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (STAPL). *4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LNCS 1511, 1998.

[8] S. Campa and M. Danelutto. A framework for orthogonal data and control parallelism exploitation. In *Proceedings of ICCSA 2004*, Springer Verlag, LNCS, Vol. 3046, pages 1295–1300, August 2004.

[9] Sonia Campa. A formal framework for orthogonal data and control parallelism handling. In *Int. Conf. on Computational Science (ICCS 2005)*, eds. V. Sunderman, D. van Albada, P. Sloot, and J. Dongarra, LNCS, Springer, *to appear*

[10] C. Lengauer, M. Aldinucci, and S. Gorlatch. Towards parallel programming by transformation: The FAN skeleton framework. *In Parallel Algorithms and Applications*, 16(2-3):87-122, Gordon and Breach (Taylor and Francis group), March 2001. ISSN 1063-7192.

[11] S. Gilmore A. Benoit, M. Cole and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *Proceedings of the Intl. Conference on Computational Science (ICCS 2004)*, volume 3038 of *LNCS*, pages 289–296. Springer Verlag, 2004.

[12] M. Aldinucci and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. *In Proc. of the 1st Intl. Workshop on Constructive Methods for Parallel Programming*, MIP 9805 pages 44–58, June 1998.