A Formal Framework for Orthogonal Data and Control Parallelism Handling

Sonia Campa

Dept. Computer Science – University of Pisa,
Viale Buonarroti 2, I-56125 PISA – Italy
campa@di.unipi.it

Abstract. We propose a semantic framework for parallel programming based on the orthogonalization of data access and control concerns by means of set of abstraction mechanisms. Such mechanisms regard the description of how data has to be accessed, the description of how data has to be computed and the description of how coupling data accesses and patterns of control. Each description is represented by an abstraction mechanism formalized through a formal semantics. The set of semantics specifications defines a method to investigate the structure of the whole application. We demonstrate how this semantics provides a formal, provable method to statically or dynamically evaluate the overall performance of the application and, eventually, apply optimization rules.

1 Introduction

From the user's point of view, one of the hardest trouble in writing parallel programs is represented by the need of expressing both data and control parallel concerns at the same time. Expressing data parallelism leads to the definition of how input data have to be accessed (e.g. sequentially or in parallel); expressing control parallelism regards the description of the functional structure of the computation, hence how each accessed item has to be processed in order to get the final result. These concerns are intrinsically orthogonal and writing a parallel application today still means keeping them together in a mixture of data accessing and control code.

Since the structured parallel programming model [8] has been assessed, many works have been proposed with the aim of supporting a higher level of abstraction in programming parallel applications and, at the same time, to get the best performance. Nevertheless, nearby the extreme cases of data-driven [1, 5] and control-driven [3, 13] approaches, the most works define a two-tier architecture model in which data accesses have to be heavily detailed on top of control pattern abstractions [6, 9] or vice versa [10, 11, 12, 14]. As a consequence, data (control) intensive applications written through control-centric (data-centric) artifacts could expose a model of parallelism difficult to describe and manipulate and, in practice, a performance that is difficult to predict or to optimize.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 772–780, 2005.

This work aims at defining a new parallel programming model semantics that exposes a set of abstractions to express data and control concerns in an independent manner. The semantics provides some primitives to couple data and control expressions giving them a precise parallel behavior and to construct the whole application. Moreover, as further goal of our approach, giving data and control semantics in an independent but composite manner will be the baseline for the definition of a set of transformation rules able to evaluate and optimize the overall performance at compile and/or at run time.

Section 2 gives an overall introduction to the model we are working on. Section 3 details the semantic formulation of the outlined model. Section 4 shows some examples of formalization and how applying some preliminary optimization rules. The section is closed with a brief outline of a first Java implementation of the semantic framework; Section 6 compares related works and Section 7 summarizes conclusions and future work.

2 The Model

The basic idea of our approach is to provide the programmer with some primitives to describe separately the two main concerns of a parallel program: data accesses and patterns of control. In particular, patterns of control are represented by compoundable *control primitives* describing the functional structure of the computation. As consequence, the application results in a graph in which the nodes represent sequential or parallel modules and the arcs represent functional dependencies among modules. Such a graph is formally described by a set of semantic operators and the framework provides an optimized implementation of the related language primitives.

On the other hand, parallel or sequential accesses on data are described by means of three mechanisms: *abstract data types*, *views* and *iterators*. An abstract data type is a representation of the raw input data that abstracts from the actual implementation and/or distribution of data. Moreover, the model provides different kind of typed views that can be declared on top of an abstract data type. Declaring a view of type v on an abstract data type a means declaring a logical structure of the raw data, so that the user can think of (and program) a as it was a v data structure, regardless its actual implementation. For example, an array view could be defined on top of a set of items evaluating it as a collection of singleton or a collection of data blocks. Further, the programmer could apply different kind of views on the same abstract data type in order to take advantage of different logics and operators to manage its data. The logical separation of input raw data and views will allow the framework to evaluate and optimize the actual implementation of an input data set and the abstract mechanisms used to manipulate it.

Each view provides a set of typed iterators on its items. An iterator is an object exposing a set of operations to get items from the view, coherently with the view type. For example, an array view provides iterators to get singleton or

block of items, while a tree view provides iterators to get subtrees, children or siblings of a given node, and so on. The behavior of the iterator can be specialized in order to get items in a whole or sequentially.

Summarizing, the framework provides control primitives able to describe the functional structure of an application and instantiations of views and iterators on the abstract data types that are able to describe the access patterns to the input data. The glue between these two separated concerns is represented by a set of collective primitives that couple each selected control pattern with one or more iterators. The internal parallel behavior of each collective depends on the specific couple control pattern-iterator by which it is constructed. For example, let us consider an application that can be evaluated applying the same function to all the items belonging to an array view. The application involves only one pattern of control (it is called `apply-to-all` and applies a function to all the items of its input in parallel) and the user can decide if instantiating an array view's iterator that returns the items in a whole or one-by-one. In the first case, combining the iterator with the `apply-to-all` control pattern will exploit a plain data parallel behavior[1] , in the second case the combination acts as a plain task parallel behavior[2].

By using such collectives and mechanisms, a programmer has independently described the computational graph and the policies to access data, leaving to the framework the optimization of the actual implementation. The optimization improvements arise from transformation rules that are statically and/or dynamically applied by the framework through the associated algebra, as demonstrated in the next two sections.

3 Semantics

Our model is completely 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 representing the raw data types, $\mathcal{V}$ is the set of view types, $\mathcal{I}$ is the set of iterator types, and $\mathcal{C}$ is the set of collective primitives.

The view abstraction is built by a set of mapping functions that, given an abstract data type $\{s_1, \ldots, s_n\} \in \mathcal{A}$ as input, returns an object belonging to the set of view types $\mathcal{V}$. At the moment, the provided view types are `Array, Matrix, Graph, Tree` and `List`. For example, given $s = \{s_1, ..., s_n\} \in \mathcal{A}$, and $g \in \mathbb{N}$ as a grain value, the `Array(s,g)` type is constructed by the mapping function

$$\mathcal{A} \times \mathbb{N} \rightarrow \texttt{Array}(\mathtt{s}, \mathtt{g}) = [(s_1, \ldots, s_g), (s_{g+1}, \ldots, s_{2g}), \ldots, (s_{size(s)-g}, \ldots, s_n)]$$

[1] Pattern called `map` in the skeleton community's jargon.
[2] Pattern called `farm` in the skeleton community's jargon.

Each view is a factory for a set of iterators on its structure. Generally speaking, an iterator is a couple (v, p) where $v \in V$ is the (sub)view on which it applies and $p \in \mathbb{N}$ is a pointer to the next accessible item of its range. Thus, an iterator is given by the function

$$\mathcal{V} \rightarrow \mathcal{V} \times \mathbb{N}$$

that constructs the couple $(v, p) \in \mathcal{I} = (\mathcal{V} \times \mathbb{N})$ on a given view, according to the factory rules of the view type. Tab. 1 shows some operations to handle an iterator object. Type $\mathcal{O}$ represents a super type for objects. The operator *curr* returns the current item which the pointer p points to. If the iterator has a parallel behavior, then 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.

Table 1. Sample operators on iterators

$curr : \mathcal{I} \rightarrow \mathcal{O}$	$curr((v,p)) = v_p$
$skip : \mathcal{I} \rightarrow \mathcal{I}$	$skip((v,p)) = (v, p+1)$
$hasNext : \mathcal{I} \rightarrow Bool$	$hasNext((v,p)) = (p \leq \#v)$
$\vdots$	$\vdots$

Using $\mathcal{A}$, $\mathcal{V}$, $\mathcal{I}$ objects and its operators allows to describe access patterns to the input data. On the other hand, it is possible to describe control patterns by means of a set of control primitives. Typically, a control primitive is a function $\mathcal{C} : \mathcal{I} \times (\mathcal{C} \cup Function) \rightarrow \mathcal{V}$ that gets a sequential function (its type is called *Function*) or a nested control primitive and a set of values (i.e. the ones given by one (or more) coupled iterator(s)) and returns the view obtained by applying the second argument to the item set. The parallel access behavior by which this item set will be accessed is encapsulated by the iterator instance given as input: this means that the programmer can concentrate on the structural aspects of his computational graph, regardless how to deal with data parallel concerns.

In our semantic model, operators on iterators and collectives (that is the control pattern concerns) are evaluated by means of inference rules that define how each of them has to be transformed by the evaluation relation $\rightarrow$.

Let us give some example of how this relation works and on which operators.

Since an **iterator** is represented as a couple $(\overline{x}, p)$ where $\overline{x} = [x_1, ..., x_n]$, the *curr* operator is in charge of returning the p-th element (*SeqIterator* type) or all the elements as a whole (*ParIterator* type). The *curr* operator is evaluated by the following rule:

$$\frac{it = ([x_1, \ldots, x_n], p) : SeqIterator}{curr(it) \rightarrow^{curr} x_p} \quad \frac{it = (x, p) : ParIterator}{curr(it) \rightarrow^{curr} \overline{x}} \tag{1}$$

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

$$a.\frac{g(x) \to y' \wedge f(x) \to x'}{h(g(x)) \parallel h'(f(x)) \to h(y') \parallel h'(x')} \qquad b.\frac{g(y) \to y' \wedge f(x) \to x'}{f(x) \parallel g(y) \to^{par} [x', y']} \quad (2)$$

The evaluation of `par` is strict i.e. before evaluating it, all its arguments have to be completely evaluated. Hence, in case of nested functions (rule 2-a), the inner functions have to be evaluated, first. Finally, the evaluation of `par` produces a new view by appending all the parallel results in a new data set(rule 2-b).

Let f be a function to be applied to all the items provided by an iterator $it = ([x_1, \ldots, x_n], 1)$ and let $v = [f(x_1), \ldots, f(x_n)]$ be the functional result it waits for. We can compute this transformation by means of an operator that gets the couple (it, f) and, depending on the type of it (parallel or sequential), applies f to all the *curr*'s of it. The inference rules describing such a "spreading" operator (called **spread**) are given by:

$$\frac{curr(it) \to [x_i | \forall i = 1, \ldots, size(it)]}{f \ominus it \to^{spr} \|_{i=1,..,size(it)}\ f(x_i)} \qquad \frac{curr(it) \to x_p}{f \ominus it \to^{spr}\ f(x_p); f \ominus skip(it)} \quad (3)$$

As it can be seen, the evaluation of the spreading operator $\ominus$ depends on the evaluation of *curr* thus, in chain, on the type of it. If it is sequential, f is applied sequentially to all the element of it by invoking $\ominus$ recursively on a successive state of it each time. Such state is provided by the `skip` operator whose evaluation rules are very intuitive. The operator "`;`" stands for the sequential evaluation of its arguments. If it has a parallel behavior, *curr* returns all the element as a whole and they need to be evaluated by the `par` operator.

Having the set of operators given above, we are now able to describe the **apply-to-all** (α) control pattern that in parallel applies f to each element of an input dataset:

$$\frac{hasNext(it) \to true}{\alpha(f, it) \to f \ominus it \parallel \alpha(f, skip(it))} \qquad \frac{hasNext(it) \to false}{\alpha(f, it) \to []} \quad (4)$$

The α operator depends on the more generic spread operator: if it is a parallel iterator, $\ominus$ will evaluate f on all the elements of it *in a whole* and the recursive call will stop the evaluation. This pattern exploits a plain data parallel behavior.

On the other hand, if it is a sequential iterator, $\ominus$ evaluates on the current singleton of it while α is invoked recursively and in parallel on the next state of it. The evaluation results in applying f to all the elements of it in but accessing the elements sequentially: this kind of pattern exploits a plain task parallel behavior.

A **pipeline** of functions can be modeled by an array or list view on the set of stages to be computed $v = [f_1, \ldots, f_n]$ and by a sequential iterator on such view, say $it_f = (v, p)$. Let it_d be a sequential iterator on the given input data set view and let $it_f \oplus x = f_n(f_{n-1}(...(f_2(f_1(x)))...))$ be the function that evaluates the pipeline of functions given by it_f on a single value x. The operator $\oplus$ (so-called **chain** operator) is evaluated through the following inference rules:

$$\frac{hasNext(it_f) \to true}{it_f \oplus x \to^{chain}\ skip(it_f) \oplus curr(it_f)(x)} \qquad \frac{hasNext(it_f) \to false}{it_f \oplus x \to^{chain}\ x} \quad (5)$$

Now we can give the inference rule able to describe a pipeline computation represented by it_f on the whole input data set accessed through the iterator it_d:

$$\frac{hasNext(it) \rightarrow true}{Pipe(it_f, it_d) \rightarrow^{pipe} it_f \oplus curr(it_d) \parallel Pipe(it_f, skip(it_d))} \quad (6)$$

$$\frac{hasNext(it_d) \rightarrow false}{Pipe(it_f, it_d) \rightarrow^{pipe} []} \quad (7)$$

It has to be pointed out that all the operators given above, are fully described by a complete set of inference rules but here we have just given as much as we consider to be sufficient.

4 Examples of Programs Evaluation

In order to show the expressive power of our formalization, we will give a simple example. Let $it_f = (< f_1, f_2 >, 1)$, $it_d = (< x_1, \ldots, x_3 >, 1)$ and $it^{(p)}$ be the notation for iterators stressing that the current state of it points to the p-th element. Moreover, let $Pipe(it_f, it_d) =< f_2(f_1(x_1)), f_2(f_1(x_2)) >$ be the expression we want to evaluate. By applying our rules, the evaluation proceeds as follows:

$Pipe(it_f, it_d)$
$\rightarrow^{pipe} \{(6)\} it_f^{(1)} curr(1)(it_d) \parallel Pipe(it_f, skip^{(2)}(it_d))$
$\rightarrow parcurr(1)(it_d)x_1, skip(2)(it_d)it_d it_f x_1 \parallel Pipe(it_f, it_d)$
$\rightarrow^{par}$ {we have to reduce `par`'s arguments, first}
$skip^{(2)} \oplus (it_f)(curr^{(1)} it_f) \oplus (x_1) || (it_f curr^{(2)}(it_d) \parallel Pipe(it_f, skip^{(3)}(it_d)))$
$\rightarrow^{par} it_f \oplus f_1(x1) \parallel (it_f \oplus x_2) \parallel Pipe(it_f, it_d^{(3)})$
$\rightarrow^{par} \{f_1(x_1) y_1\} it_f^{(2)} \oplus y_1 \parallel (it_f \oplus x_2) || Pipe(it_f, it_d^{(3)})$
$\rightarrow^{par} skip^{(3)}(it_f) \oplus curr^{(2)}(y_1) \parallel skip^{(2)}(it_f) \oplus curr^{(1)}(it_f)(x2) \parallel it_f curr^{(3)}(it_d) \parallel Pipe(it_f, skip^{(4)}(it_d))$
$\rightarrow^{par}$ {we have to reduce `par`'s arguments, first}
$it_f^{(3)} \oplus f_2(y_1) \parallel it_f^{(2)} \oplus f_1(x_2) \parallel it_f \oplus x_3 \parallel Pipe(it_f, it_d^{(4)})$
$\rightarrow^{par} \{f_2(y_1) \rightarrow y_1' \wedge f_1(x_2) \rightarrow y_2\} it_f^{(3)} y_1' \parallel it_f^{(2)} \oplus y_2 \parallel it_f \oplus x_3 \parallel Pipe(it_f, it_d^{(4)})$
$\rightarrow^{par} \{ hasNext(it_f) \rightarrow false \wedge hasNext(it_d) \rightarrow false\}$
$y_1' :: skip^{(3)}(it_f) \oplus curr^{(2)}(y_2) \parallel (skip^{(2)}(it_f) \oplus curr^{(1)} it_f(x_3)) \parallel []$
$\rightarrow^{par} y_1' :: it_f^{(3)} \oplus f_2(y_2) \parallel it_f^{(2)} \oplus f_1(x_3) \parallel []$
$\rightarrow^{par} \{f_1(x_3) \rightarrow y3 \wedge f2(y2) \rightarrow y_2'\} y_1' :: it_f^{(3)} \oplus y_2' \parallel it_f \oplus y_3 \parallel []$
$\rightarrow^{par} \{hasNext(it_f) \rightarrow false\} < y_1', y_2' >:: skip^{(3)}(it_f) \oplus curr^{(2)}(y3)$
$\rightarrow^{cons} < y_1', y_2' >:: it_f^{(3)} \oplus f_2(y_3) \rightarrow^{cons} \{f_2(y_3) \rightarrow y_3' \} < y_1', y_2' >:: it_f^{(3)} \oplus y_3'$
$\rightarrow^{cons} \{ hasNext(it_f) \rightarrow false\} < y_1', y_2', y_3' >$

In the previous transformation the *cons* (::) operator has been used for appending elements in the final list of results.

5 Transformation Rules

Once an user application has been written by means of our formalism, we could be interested in finding a semantic expression that is functionally equivalent to the user one but that exploits a better overall performance. As an example, we will refer to a classical rule involving the composition of functions presented in [4] and stating that given two functions, f and g, $\alpha((f;g),it) \leftrightarrow (\alpha(f,it);\alpha(g,it'))$ holds. In [2] it has been proved that the left-side application is more efficient than the right-side one. In the following we will show how the two-side reduction can be proved through our rules and as a consequence, how a more efficient program for the given expression can be statically found.

By applying the rules given above step by step and starting from $\alpha((f;g),it)$ we can obtain the following left-to-right transformation:

$\alpha((f;g),it)$
$\rightarrow \alpha(f;g) \ominus it \parallel \alpha((f;g), skip(it))$
$\rightarrow^* (f;g)(x_1) \parallel (f;g)(x_2) \parallel \cdots \parallel (f;g)(x_n)$
$= g(f(x_1)) \parallel g(f(x_2)) \parallel \cdots \parallel g(f(x_n))$
$= \{\text{let } it \text{ iterator on } [< f(x_1), \ldots, f(x_n) >]\}\ g \ominus it \parallel \alpha(g,it') = \alpha(g,it')$
$= \{\text{let } it \text{ iterator on } [\alpha(f,it)]\}\ \alpha(f,it);(g,it').$

On the other hand, the right-to-left side of the transformation can be easily proved by applying the same steps in an inverse order.

The semantics provided so far allows us to describe the behavior of parallel programs in a uniform manner, simply involving iterator and control pattern concepts. The advantage gained immediately is comparing program behaviors and structures: we can statically define transformations between programs that are semantically equivalent but that exhibit different performance when implemented on the target architecture. Since these transformations are well described within the semantic framework, they can be easily implemented, evaluated and applied without the programmer intervention.

Just to prove the feasibility of our approach, we have developed a first Java **??** practical environment [7] implementing the semantic framework. At the moment, it offers `Array`, `Graph`, `Tree`, `List` and `Matrix` view abstractions to be treated. Some preliminary experimental results based on matrix multiplication obtained on a single versus a dual processors architecture have shown a good scalability trend with an efficiency near to 97%.

6 Related Works

The idea of using the "iterator" concept as a means of "ranging" data is not new in the field of parallel programming. In the STAPL library [14], for example, iterators are called *prange*, they work inside distributed data structures or ("containers") and they represent pieces of data on which a given node should compute. Although the approach seems the same, with respect to STAPL our

abstract mechanisms are quite different. In particular we use views instead of containers for organizing input data and different views can be applied to the same input data. Moreover, in STAPL a semantic basis leading to a static performance analysis is completely missing.

7 Conclusions and Future Work

We have outlined a formal basis for expressing in an orthogonal, independent manner data and control concerns in a parallel program by means of separated but compoundable abstraction mechanisms and operators. With respect to our previous work in which we proved the feasibility of this approach based on iterators and primitives by scratching a first implementation framework, the main focus of this work has been the introduction of a semantics associated to both the basic abstractions and operators which leads to the formal definition and evaluation of transformation rules. We have shown how such transformation can be done in order to optimize the parallel behavior of the whole application.

Future work will address the extension of the semantics by new control patterns (i.e. the irregular ones as *D&C* or broadcast patterns) and new transformation rules related to the set of the new given operators. Moreover, we are working on a costs model associated with the transformation rules through which predicting how much each transformation costs and, as a consequence, which one of two functionally equivalent semantic expressions is cheaper, i.e. more efficient.

Also, we will support such extensions into our Java prototype.

References

1. Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.
2. M. Aldinucci and M. Danelutto. An operational semantics for skeletons. In *Proceedings PARCO'2003*, 2003. to appear.
3. F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.
4. Backus. Can programming be liberated from the von neumann style? A functional style and its algebra of programs (1977). In *ACM Turing Award Lectures: The First Twenty Years, ACM Press Anthology Series, ACM Press, New York*. Addison-Wesley, 1987.
5. Henri E. Bal and Matthew Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, July/September 1998.
6. 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.
7. 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.
8. Murray Cole. *Algorithmic Skeletons: structured management of parallel computation*. Monograms. Pitman/MIT Press, Cambridge, MA, 1989.

9. Manuel Díaz, Bartolomé Rubio, Enrique Soler, and José M. Troya. Integrating task and data parallelism by means of coordination patterns. *Lecture Notes in Computer Science*, 2026:16, 2001.
10. 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.
11. H. Kuchen. A skeleton library. *Lecture Notes in Computer Science*, 2400:620–628, 2002.
12. H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141, June 2002.
13. G. A. Papadopoulos and F. Arbab. Control-driven coordination programming in shared dataspace. *Lecture Notes in Computer Science*, 1277:247, 1997.
14. L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (STAPL). *Lecture Notes in Computer Science*, 1511, 1998.